AFRL-RI-RS-TR-2014-052

CYBER-PHYSICAL MULTI-CORE OPTIMIZATION FOR RESOURCE AND CACHE EFFECTS (C2ORES)

VANDERBILT UNIVERSITY

MARCH 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-052 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILLIAM MCKEEVER
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2014	FINAL TECHNICAL REPORT	JUL 2012 – SEP 2013

4. TITLE AND SUBTITLE

CYBER-PHYSICAL MULTI-CORE OPTIMIZATION FOR RESOURCE AND CACHE EFFECTS (C2ORES)

5a. CONTRACT NUMBER
FA8750-12-1-0245

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
63781D

6. AUTHOR(S)

Douglas C. Schmidt

5d. PROJECT NUMBER
SISP

5e. TASK NUMBER
VC

5f. WORK UNIT NUMBER
OR

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Vanderbilt University
2305 West End Avenue
Nashville TN 37203

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-052

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes the progress made on a one year project entitled *Cyber-physical multi-Core Optimization for Resource and cachE effectS* (C^2ORES). The project set out to tackle core research needed to automate and optimize the allocation of software to computing cores for cyber-physical DoD systems and create prototypical tools needed by DoD programs and developers. It yielded a system execution modeling and deployment optimization platform designed to help developers and testers of cyber-physical DoD systems choose a task parallelization and distribution architecture for a range of multi-core hardware platforms, including both homogeneous and heterogeneous multi-core hardware platforms.

15. SUBJECT TERMS

Homogeneous, Heterogeneous, Multi Core Hardware, Prototypical Tools

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILLIAM MCKEEVER**
U	U	U	UU	79	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SUMMARY

This report describes the progress made on a one year project entitled Cyber-physical multi-Core Optimization for Resource and cachE effectS (C^2ORES). The project set out to tackle core research needed to automate and optimize the allocation of software to computing cores for cyber-physical DoD systems and create prototypical tools needed by DoD programs and developers. It yielded a system execution modeling and deployment optimization platform designed to help developers and testers of cyber-physical DoD systems choose a task parallelization and distribution architecture for a range of multi-core hardware platforms, including both homogeneous and heterogeneous multi-core hardware platforms.

2. INTRODUCTION

Cyber-physical DoD systems, such as mission/flight avionics, ISR systems, or space systems, are commonly built using sensor-sampling/event-driven architectures. For example, one or more sensors (e.g., radar tracker, altitude indicator, and speed indicator) sample data at varying rates. The sampled data is then transformed into software events that are processed by event handlers (e.g., flight control software elements). Unlike traditional software event handlers (which interact with hardware/software related-events, do not receive inputs from the physical environment and are relatively deterministic), real-world dynamics directly impact the generation of software events that flow through cyber-physical systems, thereby increasing non-determinism.

Cyber-physical DoD systems have traditionally been built using single core processors, where performance prediction, verification, and validation activities are well understood. The latest processor improvements, however, largely focus on increasing the number of cores (both homogeneous and heterogeneous cores) on a chip, rather than increasing clock speeds. A key challenge with migrating cyber-physical DoD systems to multi-core architectures is understanding how real-time software tasks [1] associated with events distributed across cores impact both performance and cache effects. In particular, cache effects play a substantial role in determining whether or not cyber-physical DoD systems meet their real-time deadlines. Moreover, software developed for these systems often remains decoupled from the physical dynamics of the system. All these factors lead to significant costs in the verification and validation (V&V) process. The remainder of this section describes our method of attack to minimize these costs by automating and optimizing the allocation of software to computing cores [2] for cyber-physical DoD systems.

An open problem in the production of cyber-physical DoD systems involves predicting if the software architecture (including task parallelization/interaction and distribution of tasks to cores) will meet real-time deadlines in a given cyber-physical environment. Complex cache effects, unforeseen resource contention, physical dynamics, and layers of software abstractions coupled with a lack of open tool support make it hard to predict the performance of cyber-physical DoD systems early in system lifecycles. As a result, changes to software architecture, such as changes to the task parallelization architecture and distribution of tasks to cores (e.g., to handle new requirements stemming from an expanded threat spectrum or in response to breakthroughs in multi-core hardware) can be expensive if done late in the development cycle.

Another challenge is optimizing the software-to-hardware mapping, called a *deployment topology*, to improve QoS properties, such as execution time, memory footprint, and power consumption [3,4]. Cyber-physical deployment topologies for DoD systems must meet multiple complex constraints that cannot be managed in any single tool or technique. Instead, heuristic optimization algorithms and emulation tools must be integrated and used together to derive a deployment that is both correct and optimized for a given concern, such as power consumption. A significant gap exists in prior research for integrating multiple disparate algorithms, methods, and tools for deployment optimization and emulation into a single unified deployment optimization platform.

For example, advanced deployment topology techniques based on bin-packing [6,7,8,9,10] or evolutionary algorithms [5,11,12] are estimate software execution time. At runtime, the actual software may behave differently than expected (e.g., execute faster or slower). It is therefore critical to iteratively explore the search space with both optimization tools (to identify new candidate deployment topologies) and emulation tools (to accurately assess QoS characteristics if the system is deployed with the given topology) [13,14]. Moreover, even if a group of techniques needed to produce an optimized deployment plan can be applied, complex synchronization must be performed to ensure deployment model consistency [15,16,17,18]. For example, any real-time scheduling analysis assumptions about execution time must align with known information on cache effects of co-located components. What are needed, therefore, are highly automated, hybrid (i.e., genetic/evolutionary plus bin-packing) deployment optimization algorithms and emulation synchronization mechanisms that can account for and optimize across the multiple deployment objectives and constraints of cyber-physical DoD systems.

3. METHODS, ASSUMPTIONS AND PROCEDURES

3.1 Problem Statement

An open problem in the production of cyber-physical DoD systems involves predicting if the software architecture (including task parallelization/interaction and distribution of tasks to cores) will meet real-time deadlines in a given cyber-physical environment. Complex cache effects, unforeseen resource contention, physical dynamics, and layers of software abstractions coupled with a lack of open tool support make it hard to predict the performance of cyber-physical DoD systems early in system lifecycles. As a result, changes to software architecture, such as changes to the task parallelization architecture and distribution of tasks to cores (e.g., to handle new requirements stemming from an expanded threat spectrum or in response to breakthroughs in multi-core hardware) can be expensive if done late in the development cycle.

Another challenge is optimizing the software-to-hardware mapping, called a *deployment topology*, to improve QoS properties, such as execution time, memory footprint, and power consumption [3,4]. Cyber-physical deployment topologies for DoD systems must meet multiple complex constraints that cannot be managed in any single tool or technique. Instead, heuristic optimization algorithms and emulation tools must be integrated and used together to derive a deployment that is both correct and optimized for a given concern, such as power consumption. A significant gap exists in prior research for integrating multiple disparate algorithms, methods, and tools for deployment optimization and emulation into a single unified deployment optimization platform.

For example, advanced deployment topology techniques based on bin-packing [6,7,8,9,10] or evolutionary algorithms [5,11,12] estimate software execution time. At runtime, the actual software may behave differently than expected (e.g., execute faster or slower). It is therefore critical to iteratively explore the search space with both optimization tools (to identify new candidate deployment topologies) and emulation tools (to accurately assess QoS characteristics if the system is deployed with the given topology) [13,14]. Moreover, even if a group of techniques needed to produce an optimized deployment plan can be applied, complex synchronization must be performed to ensure deployment model consistency [15,16,17,18]. For example, any real-time scheduling analysis assumptions about execution time must align with known information on cache effects of co-located components. What are needed, therefore, are highly automated, hybrid (i.e., genetic/evolutionary plus bin-packing) deployment optimization algorithms and emulation synchronization mechanisms that can account for and optimize across the multiple deployment objectives and constraints of cyber-physical DoD systems.

3.2 Proposed Research Approach

To address the problem described above, we propose to research and prototype ***Cyber-physical multi-Core Optimization for Resource and cachE effectS (C^{2}ORES)***. C^{2}ORES is a system execution modeling and deployment optimization platform that supports continuous integration and testing, which helps developers and testers of cyber-physical DoD systems choose a task parallelization and distribution architecture for a range of multi-core hardware platforms, including both homogeneous and heterogeneous multi-core hardware platforms. Figure 1 shows how the algorithms, methods, and tools developed in the C^{2}ORES research effort will allow developers and testers of cyber-physical DoD systems to complete the following tasks:

1. ***Domain-specific modeling of cyber-physical DoD systems***. System designers model their cyber-physical platform and multi-core hardware using extensible domain-specific modeling languages that allow developers to write a single design model and generate multiple implementations optimized for specific multi-core platforms and physical environments. Domain-specific modeling languages capture the abstractions and semantics of the target domain at both the cyber-level and the physical-level, as well as the interface that tightly couples the two abstractions, which helps to model the flow of events between the two abstractions [19,20,21,22].
2. ***Deployment optimization based on cache effects***. Automated heuristic/metaheuristic algorithms optimize the allocation of software tasks to cores based on key performance criteria, such as cache effect information, real-time deadlines, memory consumption, and network bandwidth consumption.
3. ***Emulation for realistic feedback***. Given the cyber-physical platform and multi-core hardware models, C^{2}ORES will generate emulation code from the models such that it conforms to the target platform (and architecture). C^{2}ORES will then use the optimized deployment topology to map the system onto actual hardware cores and evaluate its performance. This emulation will enable C^{2}ORESto validate the correctness of the optimization algorithms under realistic operating conditions.
4. ***Continuous and iterative searching***. Steps 1–3 in Figure 1 and discussed above focus on a single deployment that optimizes cache-effects based on current information. This deployment of the cyber-physical DoD system, however, may not be the most optimal deployment, may miss real-time deadlocks, or suffer from deadlocks. C^{2}ORES will therefore

iteratively search the deployment solution space using available knowledge (e.g., what was learned from the previous emulations in a realistic environment) until cyber-physical design constraints (such as real-time deadlines, memory limits, network bandwidth consumption targets, and end-to-end execution time requirements) are met.

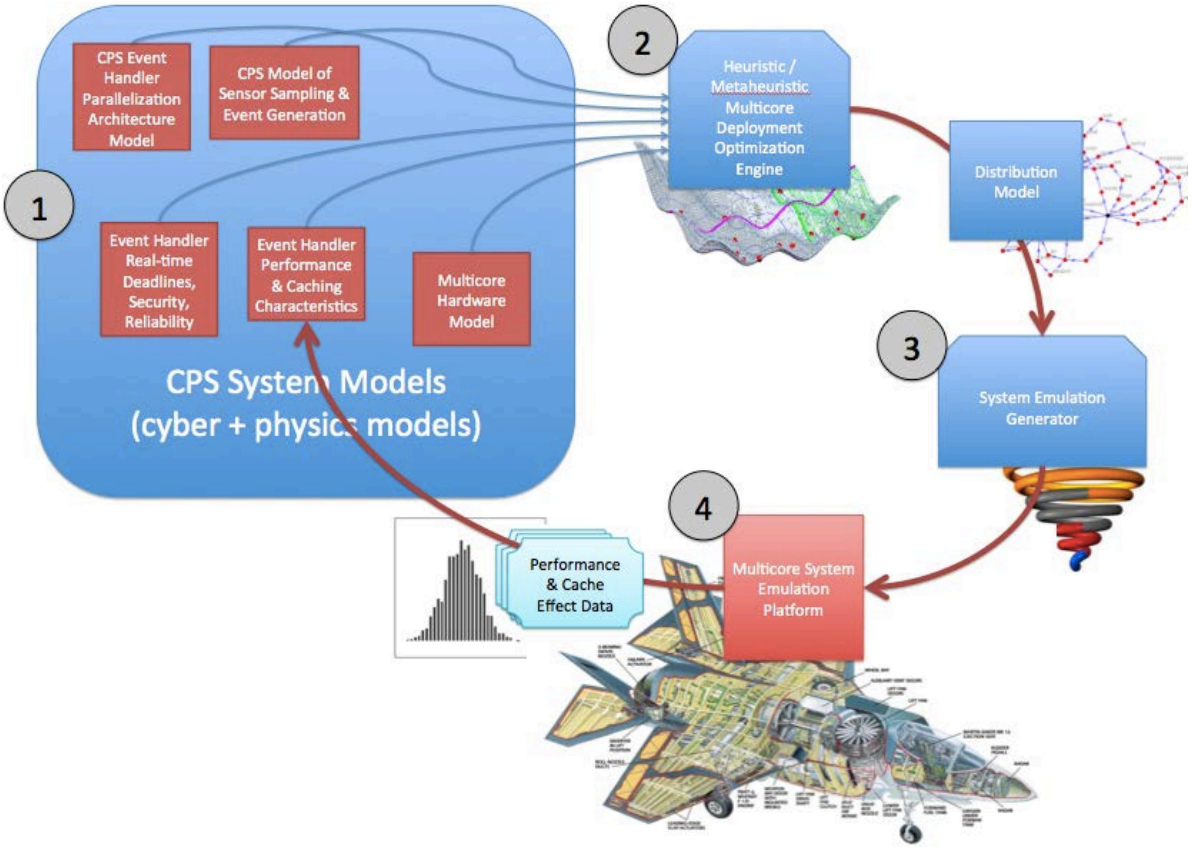


Figure 1: The C[2]ORES Workflow and Solution Approach

The proposed research described in the proposed C[2]ORES project builds upon our prior work [13,14,23,24,25,26] on cyber-physical systems that execute on multi-core architectures. An important outcome of our proposed project will be a tool-suite including model-driven engineering capabilities, heuristic/meta-heuristic algorithms, validation mechanisms based on emulation on realistic testbeds, and feedback capabilities that enable iterative refinements to the process. We will demonstrate the efficacy of our techniques and the tool suite on a number of challenge problems of interest to DoD and Air Force in particular. These challenge problems will be drawn from those listed on the OSD-sponsored SPRUCE project (www.sprucecommunity.org). We have identified the following relevant SPRUCE challenge problems for our initial set of demonstration goals:

- ***Cache management in multi-core systems***, which seeks new algorithms for coordinated core scheduling that accounts for the contention for resources by tasks running on different cores. This challenge problem will directly be addressed by our proposed research.
- ***High-utilization multi-core global scheduling analysis***, which seeks to determine if we can do better in terms of assuring that real-time deadlines are met even while exceeding the

CPU (cores) utilization beyond 50%. To that end, we will use our C[2]ORESsystem execution modeling tool suite to experimentally determine if the task allocations will meet real-time deadlines even while load the cores more than 50%. If we can empirically validate this via the emulation environment, we will develop the necessary theoretical underpinnings and mathematical models so that a sound formal basis can be established.

- ***Identify instances of incomplete or imprecise specifications in design models***, which seeks to identify instances of incomplete or imprecise specifications. This challenge problem is relevant to our proposed work in that results of system execution on the emulated platform may highlight imperfections or limitations of the models and/or corner cases where the heuristics/meta-heuristics may fail to provide adequate assurances to meet deadlines.

As part of the proposed work, our aim was that as additional challenge problems would get posted to the SPRUCE portal, we would identify the most relevant ones for this project. We also planned to leverage the DoD-sponsored ATAACK mobile cloud testbed funded through the DURIP program, which is deployed at Virginia Tech and Vanderbilt University to conduct our experimentation and evaluation.

3.3 Research Performed and Deliverables

This report describes the progress made on a one year project entitled *Cyber-physical multi-Core Optimization for Resource and cachE effectS* (C[2]ORES). The project set out to tackle core research needed to automate and optimize the allocation of software to computing cores for cyber-physical DoD systems and create prototypical tools needed by DoD programs and developers. It yielded a system execution modeling and deployment optimization platform designed to help developers and testers of cyber-physical DoD systems choose a task parallelization and distribution architecture for a range of multi-core hardware platforms, including both homogeneous and heterogeneous multi-core hardware platforms.

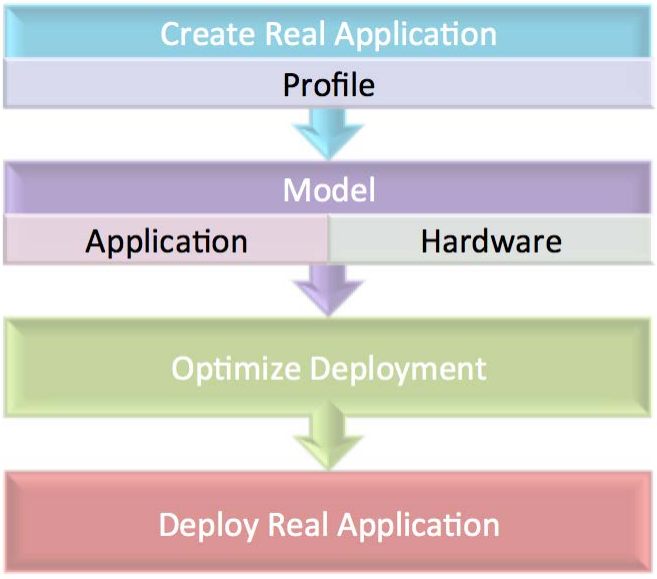


Figure 2: Project Integration Across Collaborating Teams

Figure 2 shows the workflow for collaboration across the teams involved in this project. The focus of the teams was the following:

- **Vanderbilt**: Developed models of representative cyber-physical DoD software systems & open middleware for these types of systems.

- **IUPUI**: Focused on model-driven emulation tools that performed analysis & generate code from Vanderbilt models into executable binary systems.

- **Virginia Tech**: Profiled hardware metrics, created validation methods, workflow tool-set, & documents describing inter-team APIs & dependencies.

The results of the C^{2}ORES project at the end of the first year were:

- An **integrated open-source systems execution modeling tool suite** based on model-driven engineering (MDE) technologies.

- A **methodology for evaluating and benchmarking multi-core systems** that automated and simplified many aspects of validating persistent software attributes, such as correctness and various 'ilities (e.g., scalability, predictability, and reliability).

- The results of **experiments that evaluate C^{2}ORES using the Systems and Software PRoducibility Collaboration and Experimentation Environment (SPRUCE) testbed** developed by AFRL's Software Producibility Initiative in a representative "at-scale" environment.

- **A final status report, documentation, and presentation material** that showcased the capabilities of the C^{2}ORES project.

3.4 Report Organization

The remainder of this report describes our contributions during the year-long project. The report is organized as follows: Section 4.1 provides an overview of our work on multi-core deployment optimization using simulated annealing and ant colony optimization; Section 4.2 describes our validation framework for multiprocessor and distributed scheduling algorithms; Section 4.3 describes an intelligent and tunable power- and performance-aware virtual machine placement technique for cloud-based real-time applications; and Section 4.4 describes a model-driven engineering tool that supports abstract emulation of CPU workload generation.

4. RESULTS AND DISCUSSION

4.1 Multi-core Deployment Optimization Using Simulated Annealing and Ant Colony Optimization

This work introduces a hybrid meta-heuristic algorithm for solving the problem of multi-core deployment optimization (MCDO). It extends prior work using Ant Colony Optimization to solve MCDO initially seeding the pheromone matrix with the output of a Simulated Annealing metaheuristic. This new hybridized algorithm is shown to have a median improvement in

makespan time of 7.2% across 28,800 different algorithm inputs. Moreover, there is a median reduction of 74% in execution time over the non-hybridized algorithm.

Emerging Trends and Challenges. Multicore systems are critical in modern computing. These systems are used for problem domains such as graphics computing and digital signal processing. While the theoretical processing power of these systems is quite large, there are a number of challenges to effectively utilizing all cores. Software can typically be broken into a number of separate tasks, but there are a number of constraints that make it difficult to determine which tasks should be executed on which processing cores. Multi-core Deployment Optimization (MCDO) formalizes this challenge, and aims to provide a mapping of software tasks onto hardware processors in such a way that an objective function, typically the makespan or overall execution time of all jobs, is minimized. In the general case, and even in many cases with relaxed assumptions, obtaining optimal mappings has been shown to be NP-Hard [39, 44].

Open Problem: Rapidly Creating Minimal Execution Time Deployments on Heterogeneous Processors with Communication Costs

Within the context of MCDO, there has been substantial work on addressing the challenge under a number of assumptions. These assumptions typically include simplifications such as homogeneous computing nodes, no communication delays between nodes, or an infinite supply of computing nodes. However, until recently there has been less focus on the more general version of the challenge, which relaxes the assumptions of homogeneous processors and zero communication delays. While a formal outline of the problem is provided later in this section, a brief is given here. Each software task has pre-established dependencies, as shown in Figure 3, that restrict a task from running until all of its predecessor tasks have completed. If a task A and its dependency task A' are executed on different processing cores P and P', then a message must be routed from P' to P upon completion of A'. The time required to route this message is dependent upon the two processing cores in question, as neighboring cores can communication more quickly than remote cores. Moreover, each processing core is considered heterogeneous, so some cores will execute software tasks rapidly while others will execute tasks slowly.

Solution Approach: Using Hybrid Metaheuristics to Rapidly Search the Solution Space

To address the Multi-core Deployment Optimization, we have implemented a hybrid metaheuristic algorithm that searches for a mapping of tasks onto processors which results in a minimal makespan. The algorithm continues work in the area of using Ant Colony Optimization (ACO) to solve the MCDO, but differs from prior work in that it combines an ACO with a Simulated Annealing (SA) algorithm. As shown in the experimental results, the hybridized SA+ACO algorithm not only showed an average improvement over the non-hybrid ACO algorithm across a broad range of inputs, it also showed a large decrease in running time. This research provided the following contributions to the study of Multi-core Deployment Optimization:

- A new hybridized metaheuristic algorithm that is shown to be effective on a large range of input spaces. This algorithm is provided in open source format for other researchers to experiment and modify.

- An extension of the Standard Task Graph benchmark data with the best scores achieved by our algorithms on problems that currently have no known optimal solution.

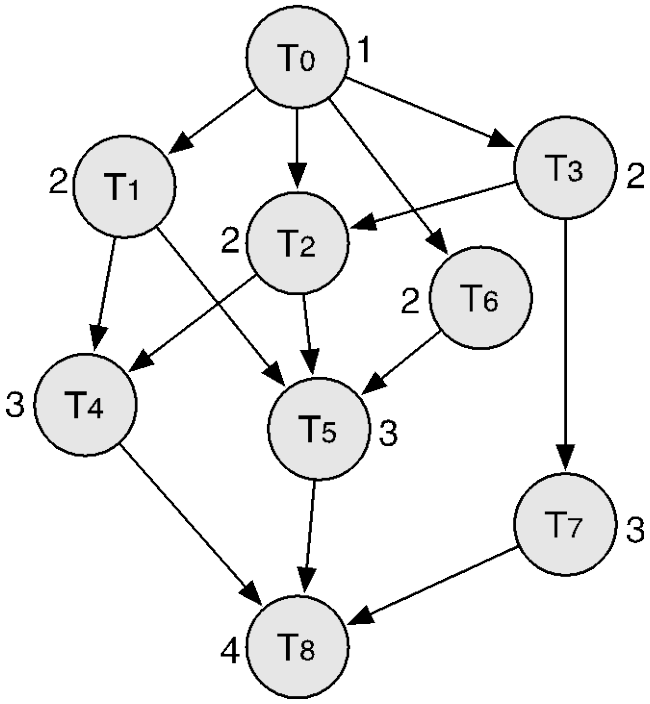


Figure 3: Task Dependency Graph for Program with 9 tasks and 4 Execution Priority Levels

4.1.1 Multi-Core Deployment Optimization

Multi-core Deployment Optimization is the challenge of mapping software components, or tasks, to hardware cores in such a way that the total execution time of the system is minimized (This is also known in the literature as multiprocessor task scheduling). For this work, cores are heterogeneous, meaning that some cores can fully execute a given task more quickly than other cores. Tasks are frequently dependent upon other results from other tasks, as shown by the Directed Acyclic Graph (DAG) in Figure 3.

This task dependency leads precedence constraints, where a task cannot execute before all of its predecessors have completed executing. Moreover, as shown in Figure 4, if a task and its predecessor are executed on different cores, then a message must be routed from the predecessor to the successor indicating the result of the predecessor's computation. This work assumes that all processing cores are connected through at least one path, although the message routing delay between two cores may be substantial.

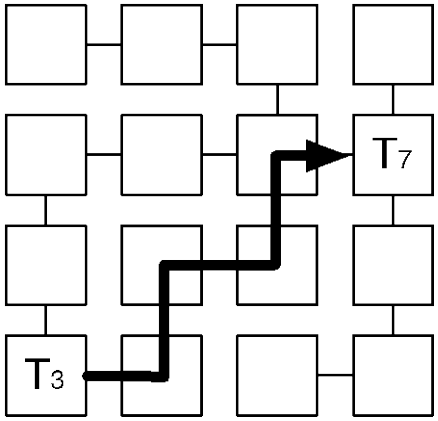


Figure 4: Routing delay of 5 hops on a fully connected processor array

4.1.2 Model of Multi-core Deployment Optimization

The Multi-core Deployment Optimization can be defined as follows:

$$D =< C, \vec{T}, O_{ij}, E_{ij}, R_{ij}, \vec{M}, sch(\vec{T_i}) >$$

where

- *C* is the set of available cores.
- *T* is the vector of tasks that need to be mapped to cores.
- *O* is a graph describing a partial ordering of tasks, where $O_{ij} = 1$ defines that task *i* must be preceded by task *j*. It is assumed that there is a single start node and a single end node to the graph, which can easily be enforced by adding two 'dummy' nodes with zero execution time.
- *E* is a matrix describing the execution time of a each task on each core. E_{ij} defines the time needed for task *i* to execute on core *j*. If all cores are homogeneous, then all columns of the matrix will be identical. If cores are heterogeneous but consistent, then all columns can be linearly transformed into one another. If cores are non-consistent, or provide specialized computing features that will make certain tasks run more or less quickly, then there is no guaranteed relationship between the columns. In this work, all cores are considered heterogeneous and consistent.
- *M* is a vector defining the mapping of each task to a core. *M*[*i*]=*j* indicates that task *i* is mapped to core *j*.
- *R* is a matrix defining the routing delay between two cores. R_{ij} defines the time to route from core *i* to core *j*. For all problems considered in this work, *R* is a symmetric matrix e.g. routing times from *i* to *j* are always equal to routing times from *j* to *i*. However, the routing delay between cores can vary drastically depending upon what physical routes link those cores together. If *O* defines that task *b* is directly preceded by task *b*', and

$M[b'] \neq M[b]$, then b cannot begin execution until b' is complete and a message containing the result of b' has been routed to the core that b is executing on e.g. $R_{M[b']M[b]}$.

- *sch(T_i)* defines, for each task, the precise order in which that task will be scheduled for execution. This is a total ordering of the partially ordered set given by *O*, and can either (depending upon the specific problem being solved) be an input to the Multi-core Deployment Optimization or an output from the algorithm. In this work, *sch(T_i)* is an output of the problem. This becomes useful in the case that two tasks incomparable in *O* are both mapped to the same core.

A valid solution to the problem passes the following constraints:

$$|\vec{M}| = |\vec{T}| \tag{1}$$

$$\forall M_i \in \vec{M}, (M_i \geq 0) \wedge (M_i \leq |C| - 1) \tag{2}$$

4.1.3 Related Work

Due to both the importance and the age of the MCDO challenge, there is a large amount of related work. The related work can be grouped by either characteristics of the exact variant of the problem being solved or by the solution approach being used. We reference both methods to ensure a reader can appropriately place this work. Common variants of MCDO include homogeneous or heterogeneous processing cores and dynamic (online) or static (offline) problems. This research falls into static scheduling, which is categorized by [46]. Other algorithms that try to address the MCDO studied in this work include earliest-time-first, modified critical path, and the localized allocation of static tasks[46].

Performing a search based upon solution approach, solutions use methods such as approximation (e.g. the depth first/implicit heuristic search[18]) and heuristic approaches(e.g. opportunistic load balancing[28]). A major subcategory of these works is list scheduling, where an ordered list of tasks is constructed by assigning a priority to each task and then iteratively scheduling tasks in this priority ordering. Genetic algorithms have been used quite heavily as a meta/heuristic approach [42].

There are a number of works that use similar problem definitions and solution approaches. The ACO algorithm used in this section is similar to the approach proposed in [49], although a less general problem model is used. [48] performs an ant colony optimization, but does not consider communication delays between processors. [27] uses a very similar ACO problem structuring, although they do not consider communication latencies. Because of this, their structuring of the pheromone matrix only focuses on job ordering, leaving processor placement as a secondary concern. [31] use a similar pheromone matrix representation but do not consider the general case of MCDO, as their work focuses on energy usage.

4.1.4 Challenges of Minimizing Task Execution Time on a Multi-core Computing System

Effective solutions to the Multi-core Deployment Optimization (MCDO) problem can vastly increase the value of many modern computing systems by increasing the throughput of the hardware without any physical modifications. With new trends towards massively parallel computer systems, MCDO a critical practical issue for obtaining the full potential of these new systems. However, MCDO has been shown to be NP-hard [32, 39, 47] by multiple researchers. Below we outline some of the specific challenges encountered while trying to find short runtime, high performance solution algorithms to the MCDO.

Challenge 1: Complexity of Multi-core Deployment Optimization Necessitates Algorithm Scalability

Multiple works have shown that MCDO is NP-Hard[32, 39, 47]. Unless P=NP, there are no methods to find an optimal solution to MCDO polynomial time. While a number of assumptions can be made to help simplify the challenge, and therefore hopefully reduce the time complexity, a combination of multiple assumptions must be made to reduce the complexity from NP-hard[47]. Adding these simplifying assumptions reduces the practicality of the final solution. Therefore a critical challenge is how to find a solution approach that can scale to solving MCDO problem sets with large numbers of cores and tasks and no assumptions. Our solution below describes how we address this challenge using a combination of simulated annealing and ant colony optimization.

Challenge 2: Interdependent Constraints Complicate the Use of Heuristics

One common approach to rapidly evaluating potential solutions to MCDO through the use of heuristics that attempt to give some guidance about which solution characteristics may lead to an optimal solution. Heuristics are typically used when constructing or permuting a valid solution, and therefore might execute thousands of times in a single optimization algorithm execution. Therefore, heuristics must be simplistic and low-overhead enough to execute quickly. On MCDO variants with assumptions that simplify the problem, such as assuming that all cores are homogeneous, this is typically feasible e.g. place task on the fastest available processor. However, as the assumptions are relaxed, it becomes more difficult to find general heuristics that consistently result in better schedules as the impact of a heuristic decision is not known until all algorithm decisions are combined and scored. To combat this complexity, the number of variables considered by a heuristics must go up, which can cause the algorithm to execute at a fraction of the original speed.

Challenge 3: Comprehensive Testing of An Algorithm for MCDO is Hard

Evaluating solutions for the MCDO an inherently difficult task[14]. Many proposed algorithms for multiprocessor task scheduling simulate the performance of their algorithms on randomly generated task graphs[30, 31, 43, 50]. Researchers frequently program their own implementation of task graph generation algorithms, which have the potential to generate non-standard graphs and mislead algorithm validation[35]. Some work has been done on creating problem sets by recording the behavior of real systems, but the most promising approaches currently are using benchmark data sets or benchmark generators [36, 51]. Due to practical difficulties, very

few researchers verify their algorithm performance on real hardware systems, making it difficult to know if the assumptions made have limited the algorithm's effectiveness. Our experimental results show how we address this challenge by using the Standard Task Graph (STG) Set as a broad-range benchmark for our applications[51].

4.1.5 The SA+ACO Hybrid Deployment Optimization

Below we describe SA+ACO, our hybrid metaheuristic algorithm for optimizing MCDO for minimal makespan times. SA+ACO is a combination of the simulated annealing (SA) metaheuristic[29, 45] and the ant colony optimization (ACO) metaheuristic[34]. Specifically, SA+ACO utilizes the Ant Colony System variant proposed by Dorigo and Gambardella, which has a reduced complexity resulting in faster runtimes[37].

4.1.5.1 Mapping Simulated Annealing into Multi-core Deployment Optimization

Listing 1 shows a simplified version of the simulated annealing algorithm. While the temperature T is high, the algorithm initially wanders the solution space searching for solutions better than the initial solution. However, as T begins to decrease, the algorithm accepts moves to worse (e.g. higher makespan) states with decreasing probability. Our full approach is slightly more complex than is shown in Listing 1, as we keep a separate variable for the globally best seen solution and the current best seen solution, and reset the search to the global best state if there has not been an improvement in the global best for 200 iterations. This allows the algorithm to make a number of poor decisions in its search for a better solution, but restarts the search from the global best state if an improvement is not eventually found.

```
1  best = initial_valid_solution ()
2  T = 3500
3  while(T-- != 0):
4      new = neighbor(best)
5      P = probability_accept ( evaluate (best ),
6                               evaluate (new),
7                               T)
8      if P > random():
9          best = new
```

Listing 1: Pseudocode for Simulated Annealing

The clear critical functions of this metaheuristic are *neighbor* and *probability_accept*. Function *evaluate* simply returns the makespan of the solution. For our work, 'bad' movements (e.g. towards a higher makespan) are allowed in a probabilistic fashion using the acceptance probability defined in Equation 3.

$$P(s, s', T) = \left\{ \begin{array}{ll} 1 & \text{if } s' < s \\ 1/1 + e^{\frac{s-s'}{T}} & \text{if } s >= s' \end{array} \right\} \qquad (3)$$

where s is the original makespan, s' is the makespan being considered, and t is the temperature of the system. The temperature starts at a high value and decreases slightly every iteration of the algorithm. Figure 5 shows that the interaction of $s-s'$ and the system temperature–higher temperatures allow moves to higher energy (greater makespan) states with greater probability than lower temperatures.

The neighbor function is defined to be an unguided random search - we choose a task i and the core that task is currently mapped to j=M[i], and assign i to any core other than j. The final output of the SA algorithm is a mapping of tasks to cores. In cases with known optimal, this mapping is typically within 30-40% of optimal.

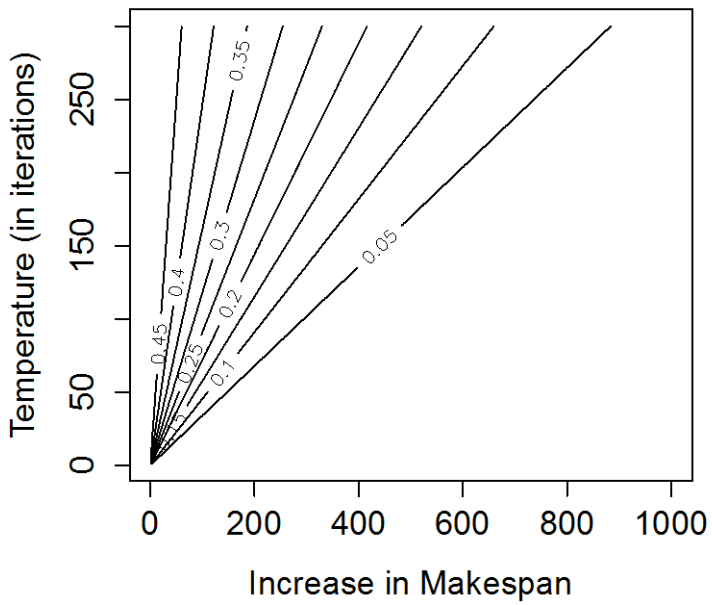


Figure 5: Probability of accepting a move to a higher energy state, a function of energy difference and current temperature.

4.1.5.2 Mapping Ant Colony Optimization into Multi-core Deployment Optimization

Ant Colony Optimization (ACO) is a biology-inspired metaheuristic that is becoming an increasingly popular approach to the Multi-core Deployment Optimization[27, 30, 31, 38, 43, 48, 50]. In general, for any problem that can be represented as a graph, ant colony optimization involves a number of 'ants' searching the solution space in a pseudo-random fashion. Each ant is searching for a complete solution, which typically takes the form of a path through the graph. When an ant locates a solution, it evaluates the goodness of the found solution and augments the pheromone matrix with information. This indirect communication through the environment, or stigmergy, allows complex overall behavior to emerge even when individual agents are simplistic e.g. memory-less. In a successful ant colony optimization the pheromone will be concentrated along routes to a good solution. To avoid local optimums, the pheromone matrix is evaporated every iteration of the algorithm, allowing more commonly-successful routes to emerge as dominant.

4.1.5.2.1 Pheromone Matrix Representation and Use

The primary decision when mapping any challenge onto an ACO is choice of pheromone matrix representation. A poor problem representation, such as one that frequently allows ants to create complete solutions without crossing the solution paths of other ants, will cause the ACO to be ineffective. While prior work has explored representing the solution matrix as task×time[48] or task×task[27], we chose to continue work that represents the solution matrix as task×core [30, 31, 43, 50], where is the affinity of task i to execute on core j. In this way, ants are searching for a tour, where a complete tour constitutes the mapping of each task onto a separate processor, that results in a shorter time when scheduled. This representation has a direct parallel to the famous shortest bridge example which shows the utility of ACO. The environmental pheromone information τ is stored a n*m matrix, where n is the number of cores and m is the number of tasks to be scheduled onto the cores. Given that the starting task was limited by O, each ant is initially deposited on a node corresponding to task 0, and (to avoid needless branching, since task 0 is known to have 0 execution time) onto core 0. From its current vertex, an ant stochastically chooses the next vertex to add to its path by balancing heuristic information and the feedback from environmental pheromones using the following formula:

$$p_{ij}^{k} = \begin{cases} \dfrac{[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in N_i^k} [\tau_{il}]^{\alpha}[\eta_{ik}]^{\beta}} & \text{if } j \in N_i^k \\ 0 & \text{otherwise} \end{cases} \qquad (4)$$

where an ant k on vertex i has a probability to add a neighboring vertex j to its path based upon the attractiveness of j divided by the attractiveness of all neighboring vertices. N_i^k is the set of neighbors of k while on vertex i. The values of α and β are parameters to the algorithm that weight the influence of the heuristic provided by η_{ij} and the pheromone trail τ_{ij}. Due to our later use of the ant colony system and simulated annealing, we chose to make the heuristic function return a fixed value.

In the Ant Colony System (ACS) proposed by [37], an additional parameter q_0 is used to directly control the rate of solution space exploration versus exploitation. The probability of adding a neighboring vertex is updated as so:

$$p_{ij}^{k} = \begin{Bmatrix} max_{l \in N_i^k}\{\tau_{il} \times \eta_{il}^{\beta}\} & \text{if } q \leq q_0 \\ \text{Equation 4} & \text{Else} \end{Bmatrix} \quad (5)$$

Therefore, q (generated from a uniform distribution of [0,1]) percent of the time, the ant will choose the best available neighbor based upon current pheromone and heuristic values. This is termed the exploration stage. If q is greater than q_0, then the algorithm will perform 'guided exploitation.'

4.1.5.2.2 Pheromone Matrix Updating

While traditional ACO performs a $O(n^2)$ update of the pheromone matrix, ACS attempts to have a faster runtime by performing all pheromone matrix evaporation while ants are already iterating the paths of their solution, either while building their solution (local update) or while depositing pheromone (global update). Only the global best ant is allowed to deposit pheromone. Equation 6 describes the global update rule.

$$\tau_{ij} = (1-p)\tau_{ij} + p\Delta\tau_{ij}^{bs}, \ \forall (i,j) \in T^{bs} \quad (6)$$

Where $\Delta\tau^{bs}_{ij}$ defined as the heuristic value of an edge that is currently on the global best solution tour. Similarly, T^{bs} is the set of all edges in the best tour. Equation 7 is executed each time an ant makes a decision while building a tour. This reduces the desirability of each edge after it is used, thereby encouraging exploration and avoiding local optimums.

$$\tau_{ij} = (1-\epsilon)\tau_{ij} + \epsilon\tau_0 \quad (7)$$

This work used 0.1 for parameter ε, as proposed by others[41]. By applying this as a local update rule, each edge that an ant used trended slightly back to the initial pheromone value τ_0.

4.1.6 Integrating SA and ACO Into SA+ACO

The hybridization of the two algorithms was done in a straightforward manner. First, the solution output by the simulated annealing algorithm was used to construct an offline ant inside of the ant colony optimization. This was done by creating an ant with a tour through the pheromone matrix that allocated each task to each core as defined in the given solution. This ant was set to be the global best solution at this point, and then an offline pheromone update was performed to seed the pheromone matrix with the initial good solution. Through experimentation it was found that further pheromone deposit on the known good solution decreased the final makespan achieved by the ant colony, and therefore we used a weighting factor to deposit 10x

the average pheromone onto the starting path. This improvement effect implies that the ant colony was being used as an effective exploitation step for improving upon the solution provided by the simulated annealing.

4.1.7 Empirical Evaluation of Learning

To address the challenge of solution evaluation, we wanted to use a standard benchmark that included both modeled and randomly-generated task graphs. The Standard Task Graph (STG) is presented by [51]. The STG is a collection of over 900 individual task graphs, including both graphs from modeled applications and from standard pseudo-random task-graph generation approaches. Moreover, each task graph has either a known optimal, or best known, solution included with the work for a range of processing cores and task counts. This allows an understanding of how far from optimal any presented solutions are, which greatly assisted during the design stages of our solution. Each task graph has a collection of associated meta information, such as the length of the critical path, which assists greatly in rapidly identifying which particular features of a graph will cause a solution to fail.

However, the STG alone was not enough input data.[28] defines important qualities for testing solutions to the heterogeneous MPTS problem. We adapted and extend this work, and the full input space is defined as follows:

- *Core heterogeneity* - The variability present in each core. For all of our trials, cores begin as homogeneous. If core heterogeneity is not 1 for this trial, then a random number is drawn from [1,coreheterogeneity] for each core, and the execution time of all tasks on that core increases by this multiplicative factor. Values used in the experiments include [1,2,4,8,16].
- *Route heterogeneity* - Continuing the generalization described in [28], we define this parameter as a method of increasing variability in the routing delay between two cores. This parameter was eventually found to be one of the most critical parameters for both absolute makespan times and for percent improvement. Future works should explore this parameter more, and potentially use the NSEW structure of the compute grid to place an upper bound on the route heterogeneity depending upon the number of cores in the system. Values used in the experiments are [1,2,4,8,16].
- *Route default cost* - This parameter was used to set the default routing cost, which was then multiplied by random numbers drawn from a uniform distribution of [1,routeheterogeneity]. Setting this parameter to zero allows the SA+ACO to be executed under the assumption that there are no communication delays in the system.
- *Task count* - This parameter was retrieved from the STG graphs, and used values were [50,100,300,500,750,1000]
- *Core count* - Values used were [2,4,8,16]

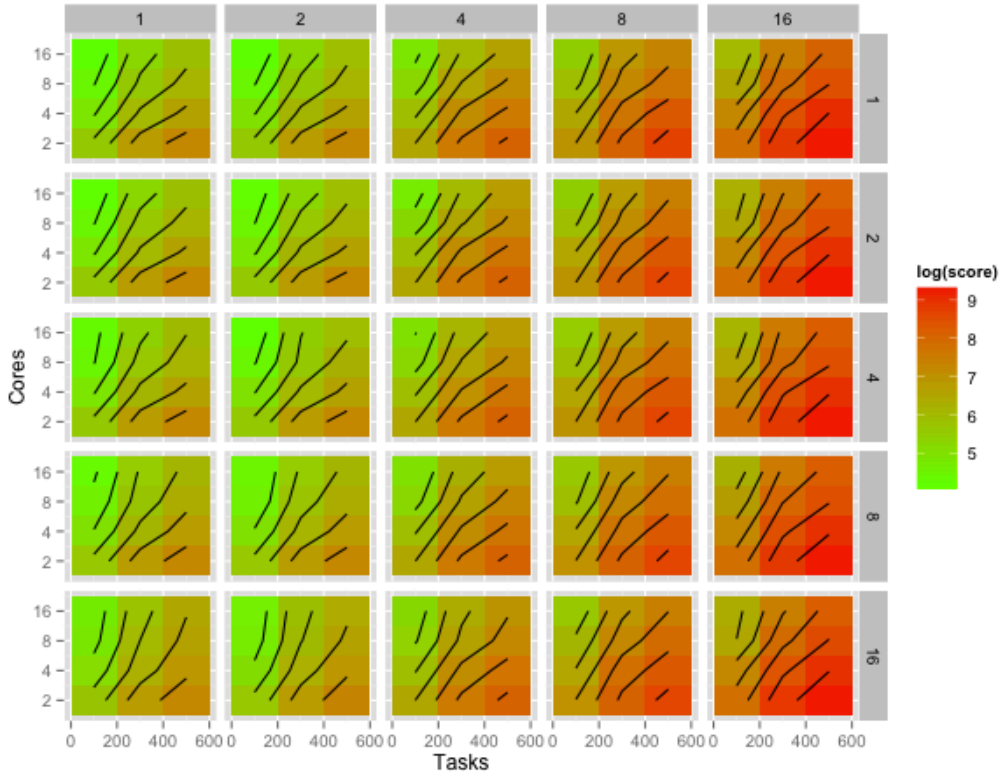


Figure 6: Makespans achieved by SA+ACO for different combinations of input parameters. Each contour line indicates a rise of 0.5 in the log(makespan)

Additionally, both algorithms used α=2, β=0.85 and q_0=0.5; ACO used ants=60, iterations=100; SA+ACO used ants=20, iterations=25, T_0=5000. Figure 6 gives some indication of the expected makespan for each input parameter combination. A similar figure for ACO looks nearly identical. Note that the route heterogeneity is clearly the most critical parameter for short makespans. There is not a significant increase in makespan as core heterogeneity increases, but we do see an expected rise in makespan as task counts increase and core counts decrease. Note that all makespan increases are on a logarithmic scale, the actual values of makespans found in Figure 6 range from 27 time units to 62,922 time units.

While we considered using the task heterogeneity metric defined in [28], the main point of the metric was to vary the distribution of task execution times. In the STG, there was a wide array of task execution times already inherent in the dataset, and taking enough samples accomplishes the same end result.

Unless stated differently in the experiment, all experiments were run on the extended dataset, which is described as so: To generate a broad testing set that could be used in a number of experiments, we took 288 graphs from the STG dataset (48 graphs from each of the 6 task counts provided by the STG). Each graph was tested using all permutations of core count, core heterogeneity, route heterogeneity, and with a routing default cost of 1.

4.1.7.1 Experimental Platform

Due to the nature of the problem under test, there is a large desire to test exhaustively and understand precisely which conditions the provided algorithm is effective for. To enable this large-scale testing, the experimental platform was a cluster of thirty-two computers and a master node. The proposed algorithm was encapsulated as a stand-alone C++ executable, and the python jug[33] framework was used to assist in running multiple iterations of the solution concurrently on the clustered computers. All C++ was compiled with g++ 4.4.6 using the -O3 flag. To assist other researchers, our exact experiment data, testbed code, and solution code will be made open source.

The master computer has an 8-core Intel Xeon X3470 2.94Ghz processor and 8GB memory. It is running CentOS 6.0, python 2.6, and jug 0.9.2. Each homogeneous clustered computer uses a Dell D610 blade mesh with two Intel Xeon E5646 12-thread processors, for a total of 24 hardware threads per computer. Each computer has 36GB of memory and is running CentOS 6.0, python 2.6, and jug 0.9.2. Jug was configured to use a filesystem (network file system (nfs)) backend for locking and task synchronization.

4.1.7.2 Experiment 1: Comparing To Known Optimal

The base STG dataset (e.g. homogeneous cores and no communication delays) included known optimal scores for all four core counts used. Comparing both the nonhybrid ACO and the hybrid ACO+SA algorithms to known optimal values will give an understanding of how well these algorithms fit into other work in the general area of MCDO. The primary point of this experiment is to understand if the algorithms perform well on the simplified case of MCDO, and to compare both algorithms to a know baseline of across a wide set of experimental data. We used Equation 8 to compare the algorithms score to a known optimal value. In all cases, Equation 8 will be negative or zero as the algorithm makespan will be greater than or equal to the optimal makespan.

$$PI(score, opt) = \frac{opt - score}{opt} \tag{8}$$

For both ACO and SA+ACO, a total of 1152 independent tests were executed. Specifically, there were 48 task graphs used for every permutation of input parameters:

- Core Count: 2, 4, 8, 16
- Task Count: 50, 100, 300, 500, 750, 1000

The parameter input values used in this experiment were route heterogeneity = 1, core heterogeneity = 1, route default = 0.

Prediction: Both SA+ACO and ACO will perform within 20% of Optimal. Our hypothesis is that both algorithms will perform within 20% of optimal on this data set. Given that the goal is to use these algorithms on presumably harder problems, where the assumptions made this this experiment are relaxed, the algorithms should perform within a reasonable margin of opti-

mal on this challenge.

Experiment 1 Results. Figure 7 shows a density distribution of the percent difference from the known optimal score for both ACO and SA+ACO. Note that there is no separation of task size or core size, so the distribution spans both the simplest (low task count, low core count) and the hardest (high task count, high core count) instances of the problem.

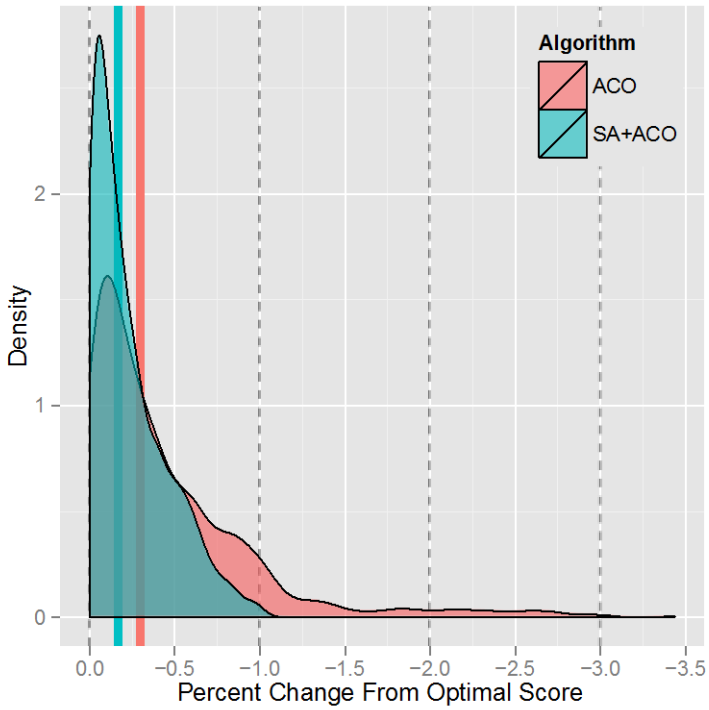


Figure 7: Comparison to known optimal values made by using Equation 8. Thick vertical lines indicate data median. 1152 samples per Algorithm.

Figure 7 shows that the median of both SA+ACO and ACO is greater than -0.5, meaning that both algorithms were able to achieve scores that were less than 50% worse than the known optimal values. However, SA+ACO clearly has a shorter tail on the graph, in the worse case creating makespans twice as long (-1.0 percent change) as the optimal makespan, whereas ACO occasionally creates makespans that are 3.5 times longer than the optimal value. The median value of SA+ACO is 16.5%, while the median for ACO is 29.5%. This slightly disproves our hypothesis, and indicates that ACO is potentially not as effective as expected. Figure 7 can be misleading in that a small change in absolute execution time can result in a large difference in percent change. Table 2.1 helps to clarify the results of Figure 7 into absolute difference in time units (e.g Equation 8 times the optimal score). Table 2.1 shows that in all subsets considered SA+ACO outperforms ACO.

Table 1: Comparison of Median Achieved Scores. Task sizes included are [50,100,300,500,750,1000]

Cores	Median OPT	SA+ACO Median	ACO Median
2	1569.5	+22	+62
4	798	+91	+217.5
8	489	+158	+313
16	343	+121.5	+317

4.1.7.3 Experiment 2: Runtimes of ACO and SA+ACO

While initial results show that SA+ACO is outperforming ACO on the version of the problem with the most assumptions, a critical question is how much extra runtime is needed to achieve these better scores. An algorithm that exhibits exponential runtime for linear increase in scores may rapidly become impractical. Equation 9 is used to determine percent change in runtime from ACO to SA+ACO.

$$PC(saaco, aco) = \frac{aco - saaco}{aco} \tag{9}$$

Prediction: SA+ACO will run substantially faster than ACO. Our initial experimentation found that the execution time of a single ant in the ACO algorithm was quite substantial. We hypothesis that this is due to an ant needing to iteratively construct a valid solution, making decisions at each step. In contrast, SA+ACO can rapidly permute an existing solution into another solution. By using SA+ACO to seed the ant colony, we can reduce the number of iterations and ants in SA+ACO, which we predict will result in net performance gains.

Experiment 2 Results. Figure 8 shows that SA+ACO is consistently faster than ACO in terms of absolute runtimes. SA+ACO finds solutions to all problems within 100 seconds(and typically in under 10 seconds), whereas ACO finds requires up to 300 seconds to find solutions. However, this can be misleading as Figure 8 does not associate the runtimes of SA+ACO and ACO based upon the exact input graph and parameters used. Figure 9 addresses this by associating all runs and calculating a percentage improvement. As shown in Figure 9, SA+ACO has a runtime that is consistently 60-90% faster than ACO.

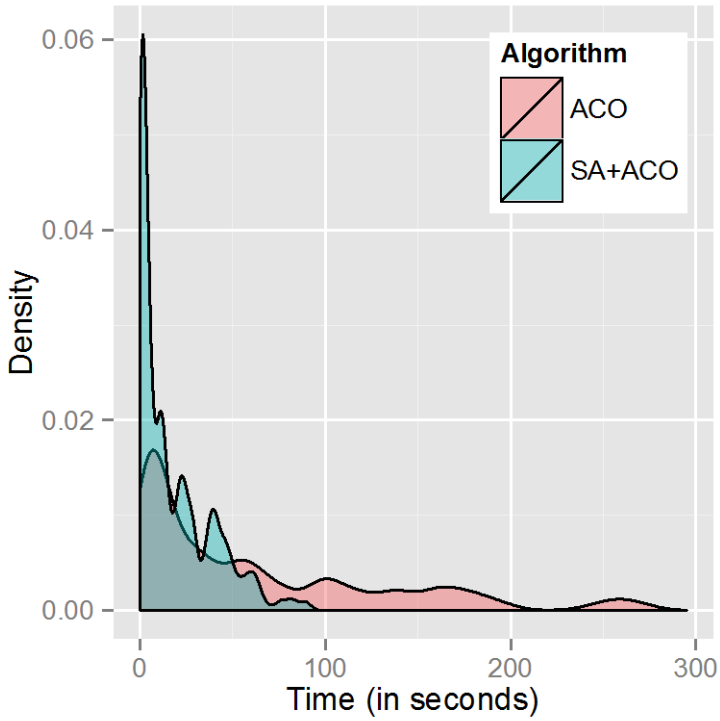


Figure 8: Density distribution of runtimes of SA+ACO and ACO.

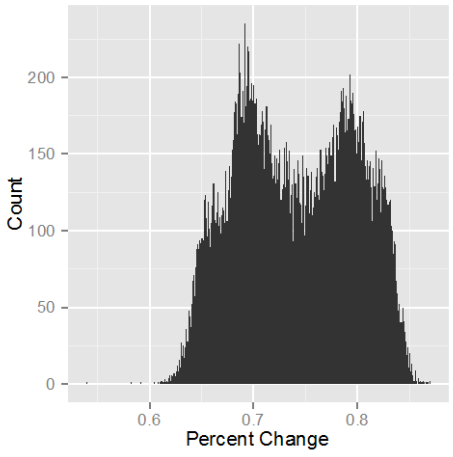


Figure 9: Density distribution of percent improvement from ACO runtime to SA+ACO runtime.

4.1.7.4 Experiment 3: Score Comparison of SA+ACO and ACO

Below we compare the makespans found by SA+ACO and ACO across the entire input space. There were 600 unique parameter combinations of core count, task count, route heterogeneity, and core heterogeneity. For each parameter combination, we performed trials using 48 different

graphs from the STG, resulting in a total of 28800 trials per algorithm. The percent difference between SA+ACO scores relative to ACO scores is calculated using Equation 9.

Prediction: SA+ACO will perform better than ACO for every combination of input parameters. Based upon the improvements made to ACO, and some initial trials, we hypothesize that SA+ACO will consistently find shorter makespans than ACO.

Experiment 3 Results. Figure 10 is a comparison of the performance of SA+ACO to ACO across a wide range of input space. For utility in discussing, we adopt the following notation: ${}_{tc}{}^{cc}x^{rh}{}_{ch}$ where cc = core count, tc = task count, rh = route heterogeneity, and ch = core heterogeneity. Having a wildcard * in any of the locations references all versions of that parameter. Having two numbers separated by a colon indicates an inclusive range. For any precise combination ${}_{j}{}^{i}x^{k}{}_{l}$ the value shown in Figure 10 is the median of 48 samples, where each sample is a different STG graph.

One of the most promising observations is the consistent improvement in makespans at ${}_{300:1000}{}^{16}x^{1:2}{}_{*}$. This case of relatively low route heterogeneity and a large number of cores and tasks is exemplary of current hardware trends in large scale computing. The effect is fairly independent with respect to core heterogeneity, meaning that slight differences in core execution speeds should not disrupt the improvement. Large computing grids with highly connected clusters are used to achieve speedups on parallelized algorithms with huge numbers of tasks, with the goal being linear speedup in the number of cores.

One of the least promising observations is at ${}_{*}{}^{8}x^{16}{}_{*}$, where SA+ACO consistently performs worse than ACO. Unfortunately, multiple attempts to dissect this failure into concrete reasoning have been unfruitful, so more work is required. One thought is that the exact combination of 8 cores and 16 route heterogeneity is probabilistically poor–there are enough routes (8*7) that multiple routes receive a high routing delay, but there are not enough total routes to move onto cores that avoid these bottlenecks. When reaching 16 cores, the number of routes increases drastically (from 42 to 16*15=240), and therefore poor routes can be avoided. Naturally this hypothesis needs to be examined before it the problem can be definitively identified. The choice of route heterogeneity = 16 may be a poor input parameter choice that is changed in future work.

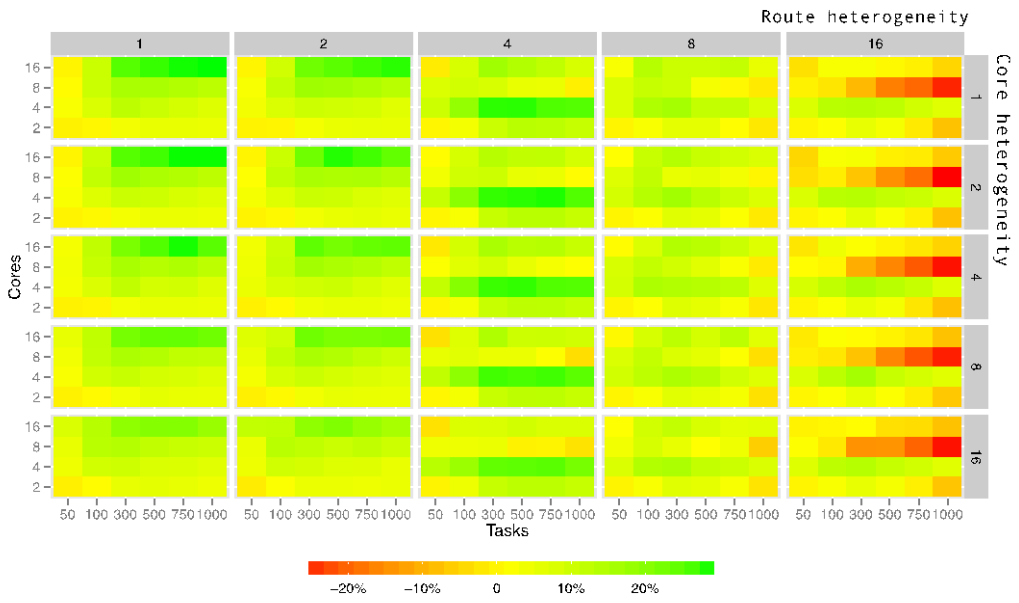


Figure 10: Percent Change from ACO score to SA+ACO score across the entire range of input parameters.

An interesting effect is noticed at ${}_{50:100}{}^{2}x^{1:2}{}_{*}$ where SA+ACO performs slightly worse. In these situations of small core counts and small task count, it seems that the ACO results in a better overall score. One hypothesis is that these problems are small enough to avoid major influence from the routing delay or core heterogeneity–this is corroborated by Figure 6 showing that task counts of 50 and 100 have consistently lower makespans across a wide range of core heterogeneity, with the makespan of ${}_{50:100}{}^{2}x^{\delta}{}_{*}$ increasing as δ increases from 1 to 16. In situations where heuristics have greater utility e.g. the effect of a adding a node to an ant's tour can be effectively predicted, the plain ACO performs better than the more random SA+ACO. In general, it is safe to assume that SA+ACO should not be the first choice in situations where the task count is 50 or less, or the core count is 2. While it will work in these situations, it will not work as well as other algorithms. We also note that SA+ACO does not perform well in ${}_{1000}{}^{2}x^{16}{}_{*}$, which falls under the rule that SA+ACO should not be used when the core count is 2.

A large performance increase is noticed at ${}_{*}{}^{4}x^{4}{}_{*}$, where SA+ACO is achieving makespans of 10-20% better than ACO. The increase continues, but at a reduced rate of 0-10%, into ${}_{*}{}^{4}x^{8:16}{}_{*}$, indicating that for 4 cores and medium to high route heterogeneity SA+ACO is a good choice.

Figure 10 is poor for directly comparing SA+ACO to ACO in a broad sense. Figure 11 provides this, showing that SA+ACO does consistently outperform ACO. The median improvement is 7.2%.

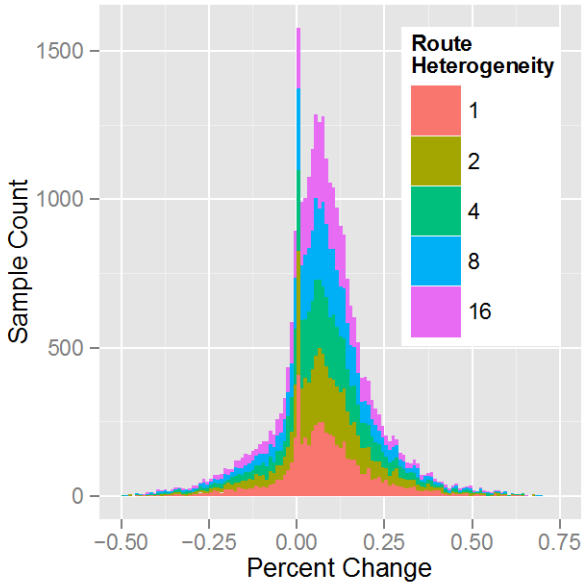


Figure 11: Distribution of percent change from ACO score to SA+ACO score.

4.1.8 Concluding Remarks and Lessons Learned

This section has introduced a new hybrid algorithm for solving the Multi-core Deployment Optimization(MCDO) problem. We have continued prior work on using ant colony optimization (ACO) to solve the MCDO, specifically by relaxing some of the assumptions of prior work and by adding an initial simulated annealing (SA) step. The two metaheuristics are interleaved, with the SA algorithm providing a seeding step for the pheromones matrix used inside the ACO algorithm.

Comparison to known optimal solutions to the MCDO is possible for a subset of the input space due to the work by Tobita and Kasahara [51] in publishing the Standard Task Graph (STG) Set, a dataset of concurrent application task execution time and precedence relations. Across 50 different concurrent applications and up to 16 cores, the hybrid SA+ACO algorithm is shown to acquire makespans that have a median of 16.5% increase over the optimal solution. The nonhybrid ACO algorithm has a median increase in makespan of 29.5%. However, optimal values are only available under a number of assumptions, such as homogeneous cores and no routing delays.

Relaxing these assumptions, we are only able to compare the performance of SA+ACO directly to the nonhybrid ACO algorithm. With one exception, across all combinations of the input parameters (core count, task count, core heterogeneity, routing heterogeneity) SA+ACO is able to find makespans that are 0% to 30% better than those found by ACO alone. Moreover, the runtime of the SA+ACO algorithm is typically 1/3 or less of the runtime of the ACO algorithm. The SA+ACO algorithm is shown to generate solutions 25% worse than nonhybrid ACO algorithm for a small class of input spaces where the routing heterogeneity is quite high (e.g. up to 16x difference in the routing time of two cores) and the number of cores is 8. More work is required to determine the exact cause for this isolated failure.

Based upon our work with these algorithms, we have created a dataset that extends the STG by detailing the makespan scores of SA+ACO and ACO for all relaxed assumptions. This extended dataset will allow other researchers to directly compare their work to ours, and provide a baseline for future attempts to solve MCDO using relaxed assumptions.

From our research on Multi-core Deployment Optimization, we learned the following important lessons:

1. Using Simulated Annealing to Seed Ant Colony Optimization is Quick and Effective. The simulated annealing algorithm described in this section eventually cools into a greedy local search. Constructing valid solutions in a local search is typically much quicker than running an iteration of an ant colony solution, as the local search simply permutes valid solutions into other valid solutions instead of creating valid solutions. However, the ant colony optimization we define is more effective than random-search simulated annealing at reusing information and performing guided exploitation of a solution space. The combination of these two metaheuristics is shown to be highly effective and worthy of continued research. A future direction might consider running a random restart simulated annealing and seeding the ACO pheromone matrix with the best N randomly found paths.
2. Routing Heterogeneity is Critical to Successful Shortest Paths. The results in this work show that routing heterogeneity is critical to shorter makespans. However, the open-ended assumption of arbitrary routing heterogeneity may be too general to be practical. Modern computer systems are often connected in a 2D plane, where each core has a NSEW connection to its neighbor. Using taxicab distance and assuming that each hop from neighbor to neighbor introduces a fixed routing delay, the interaction of routing heterogeneity and number of cores can be determined. For example, if all cores are known to contain all four NSEW connections, then the maximal routing heterogeneity is simply the largest taxicab distance. If cores are arranged in a square grid, this taxicab distance will typically equal 2*(sqrt(cores) -1), and the maximal routing heterogeneity can be fixed to this value.
3. With Relaxed Assumptions, Heuristic Creation Becomes Harder During the execution of an ant colony optimization, the heuristic function will typically be executed hundreds, if not thousands, of times each time an ant creates a tour. This requires that the heuristic function execute quickly, but this requirement for rapid execution can be difficult to achieve in highly interconnected solution spaces. For example, considering faster processors to be more desirable can leave out information about routing delays, ending in an overall adverse effect on solution. Future work should explore statically calculated data structures shared by all ants to help to enable rapid heuristic calculation, such as summary values for a core's routing centrality and a task's need for routing centrality.

The SA+ACO code, the benchmarking code (using the python jug framework), and the extended dataset containing all of our results are available in opensource form from http://www.magnum.ece.vt.edu/.

4.2 A VALIDATION FRAMEWORK FOR MULTIPROCESSOR AND DISTRIBUTED SCHEDULING ALGORITHMS

This work examines the challenge arising from the practical use of Deployment Optimization Algorithms, and primarily focuses on the need for validation of potentially beneficial algorithms. Deployment Optimization Algorithms are appropriate for optimizing many distributed and multicore systems, but even well-known algorithms can have unclear results when used in a production system due to the tendency of researchers to use a simplified model of computation when evaluating algorithms. Unexpected challenges, such as bus contention, are frequently not considered when evaluating Deployment Optimization Algorithms, which can lead to unexpected results in production environments. This section presents DOVE, a framework for validating Deployment Optimization Algorithms on real systems.

Open Problem: Validation of Deployment Optimization Algorithms. While Deployment Optimization is an active research topic, there are few real-world case studies of using the algorithms. Moreover, system designers interested in using a Deployment Optimization Algorithms cannot be guaranteed that the performance of that algorithm on their production system will equal the performance shown in the literature. Deployment Optimization Algorithms performance is not measured on real systems, but is instead is measured using an approximation of expected runtime that is composed of expected computation and routing delays. This approximation does not take into account issues such as memory size limits, routing contention, or disk I/O delays. Such issues arise can cause significant variation between the expected and actual performance of a Deployment Optimization Algorithms.

Solution Approach: Automation of Deployment Optimization Algorithms validation. To address the challenge of validation of Deployment Optimization Algorithms s we have create the DOVE framework, which automates the process of ensuring that the reported improvements in the objective function are consistent with the profiled improvements seen in a physical system. A physical hardware system is profiled to provide computation and routing delays to a Deployment Optimization Algorithms, and the full set of {solution, objective function value} pairs is then executed and profiled on a physical system.

This work provides the following contributions to the study of Deployment Optimization Algorithms:

• DOVE, a framework for automating the validation of Deployment Optimization Algorithms

• Extensive discussion on validation of Deployment Optimization Algorithms

4.2.1 Challenges of Validating Deployment Optimization Algorithms

Deployment Optimization Algorithms are an active area of research precisely because they have the potential to cause substantial performance improvements without incurring additional physical hardware costs. However, there are a number of challenges to practical use of a Deployment Optimization Algorithms, such as the inability to know the precise effects of applying

a Deployment Optimization Algorithms to an existing production environment. Below we describe some of the fundamental challenges to validating that a Deployment Optimization Algorithms algorithm can actually confer real-world benefits equal to the predicted improvements.

A. Challenge 1: Determining if Deployment Optimization is Improving Production System Metrics is Hard. Deployment Optimization Algorithms use an objective function to choose which deployment plan is most appropriate. However, there is no guarantee that the value of the objective function calculated by the Deployment Optimization Algorithms will equal the value of the objective function when executing the real system. There is little confidence that the objective function on the real system will equal the objective function value predicted by the Deployment Optimization Algorithms, due to the Deployment Optimization Algorithms operating on a simplified model of the real system that does not considering all complications, such as bus contention. While Deployment Optimization Algorithms have been shown multiple times to have beneficial effects [46], the exact amount of the effect can vary widely between each system execution.

B. Challenge 2: Identifying Reasonable Computation Limits for Optimization is Difficult. Deployment Optimization Algorithms are typically iterative algorithms, but it is unclear how many iterations are needed to find the best deployment plan without wasting compute time on the optimization. Many Deployment Optimization Algorithms include randomness, which implies that if they are allowed to run long enough they will find the optimal solution, although the algorithm user has no knowledge that this is in fact the optimal solution– extra computation time can be wasted trying to find a better solution! It is hard to predict a reasonable termination condition for any Deployment Optimization Algorithms. Moreover, algorithm performance is drastically problem dependent, and a reasonable termination condition on one problem may perform poorly on another.

4.2.2 The Deployment Optimization Validation Engine (DOVE)

DOVE enables automated validation of deployment optimization algorithms. Figure 12 shows the major components of DOVE. To validate a Deployment Optimization Algorithms, DOVE requires a physical hardware system and a software model, which contains information about data flow and component execution times for the software system being deployed. As shown in Figure 12, the hardware system is first profiled by DOVE to build an accurate description of available computation units and the routing delays between them. The model of production system software is obtained from a file in the Standard Task Graph (STG) format, which includes a directed acyclic graph of task dependency and exact execution times for each component.

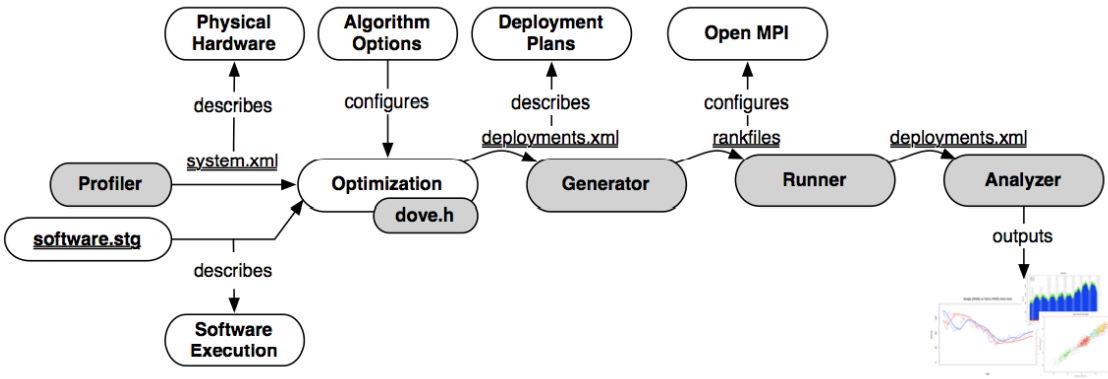


Figure 12: Overview of DOVE framework. Grey components are provided by DOVE. Underlined components indicate files

A. Profiler. The profiler tool builds an XML file describing the hardware system that will later be used for validation of a Deployment Optimization Algorithms. First, the Portable Hardware Locality tool provided by Open MPI is used to gather details on the available hardware components e.g. processors, cores, and hardware threads. Next, a latency calculation is performed between all hardware components pairs at the same level (i.e. core-core, processor-processor, etc). Most Deployment Optimization Algorithms s reviewed used a single number to represent the routing delay between processing units, and therefore DOVE's profiler runs hundreds of thousands of repetitions and returns the average routing time. Final profiler output is shown in Listing 1.

```
<nodes>
<node ip="10.0.2.4" id="0" pindex="1">
 <socket id="1" pindex="1" speed="2.4GHz">
   <core id="2" pindex="0">
     <pu id="3" pindex="0"/>
     <pu id="4" pindex="12"/>
   </core>
   ... etc
</nodes>
<routing_delays prefix="ns">
  <d f="2" t="5"  v="1438" />
  <d f="2" t="11" v="1521" />
... etc
```

Listing 1. System.xml output from DOVE Profiler

B. STG Software Model. Research in software profiling enables translating real software systems into representative formats such as STG [53],[54]. Using an intermediate format instead of using running real system code has multiple advantages: 1) by generating deterministic executable code from the software model, any variation from expectation can be attributed to errors in the hardware model, 2) any complexities with building, installing, or running a production environment can be hidden from DOVE 3) it becomes simpler to generate testcases based on common software patterns, as evidenced by [46], [51]. However, this also implies that

any performance output metrics generated by DOVE cannot be assumed to be the exact expected behavior on a production system. If the STG software model is similar to the real system, then the metrics may be the quite close, but there are some software systems, such as web services, where the resource requirements are dynamic and therefore cannot be effectively modeled using the STG format. Future work on DOVE will explore alternative methods of modeling software.

C. Optimization Algorithm Modifications. DOVE is designed to require only minimal modifications to the actual optimization algorithm. All algorithms are assumed to use an optimization model similar to the one described above, and DOVE provides functions to enable any optimization to be validated. Namely, request hardware components (i.e. allocate 12 cores) for use in a validation, get routing delays between hardware components, and store a deployment plan plus the associated model metrics. Typical inclusion would be <10 lines of additional code, as shown in Listing 2.

```
solution sol = problem->get_best_in_iteration();
dove::deployment deployment = dove->
    get_empty_deployment();
for (solution::iterator it = tour.begin();
    it != tour.end();
    it++) {
        deployment.add_task_deployment(it->task,
                                       it->core);
    }
deployment.add_metric("makespan", sol->makespan);
validation->add_deployment(deployment);
```

Listing 2. Primary code required to integrate DOVE into an algorithm

DOVE reads in the system.xml file shown in Listing 1 and provides the required information to the optimization. Currently DOVE uses a random strategy for selecting hardware components, but future work will likely use the strategy design pattern to enable strategies such as *large variance, distinct clusters,* or *low latency.* If an optimization algorithm requests N processing units of type T, then those units are referred to by the logical indices [0...N − 1] and DOVE internally translates between [0...N − 1] and the indices found in the system.xml file. An example output deployment is shown in Listing 3.

```
<optimization name="Ant Colony">
  <deployments>
    <deployment id="0">
        <deploy t="0" u="2"/>
        <deploy t="1" u="11"/>
        <deploy t="2" u="21"/>
        <deploy t="3" u="17"/>
        <deploy t="4" u="27"/>
        <metric name="makespan" value="7.8988e-05"
            unit="seconds"/>
    </deployment>
</optimization>
```

Listing 3. Deployment.xml file which represents the iterative solutions found

D. Executable System Generator. The generator included in DOVE combines a) the deployment plan output by the optimization algorithm b) the system.xml file used to find the machine-specific identifiers of all hardware components, and c) the software model. These three components are combined into an executable and a set of rankfiles (e.g. configuration files) that can be used with Open MPI to execute the software model using the deployment plans produced by the optimization. Listing 4 shows an example of the executable software code generated for Open MPI. Open MPI is configured to wait on all predecessor tasks in an asynchronous manner. Then, a busy wait loop is executed for the amount of time equal to the task's computation. Finally, all successor tasks are notified asynchronously. In Listing 4, debug output is enabled, but production code for profiling would not have the print statements.

```
case 3: {
  cout << "3: Awake" << endl;
  mpi::request req[2];
  req[0] = world.irecv(1, 0);
  req[1] = world.irecv(2, 0);
  mpi::wait_all(req, req + 2);
  cout << "3: Recv all predecessors"
    << endl;
  timespec start, end;
  cout << "3: Started compute" << endl;
  clock_gettime(CLOCK_MONOTONIC, &start);
  do
    clock_gettime(CLOCK_MONOTONIC, &end);
  while (diff(start, end).tv_nsec < 6000);
  cout << "3: Finished compute" << endl;
  mpi::request sreq[1];
  sreq[0] = world.isend(4, 0);
  mpi::wait_all(sreq, sreq + 1);
  cout << "3: Notified successors"<< endl;
}
```

Listing 4. Sample of software implementation generated by DOVE

E. Implementation runner. The implementation runner loops over each of the solutions output by the Deployment Optimization Algorithms, and times the execution of that solution. To avoid overly long profiling, the kbest algorithm is used to terminate profiling of one solution once the top k scores have converged to a threshold range, such as 500 ns. After profiling each deployment, the implementation runner updates the deployment XML file to include the total time required to execute each department. Each deployment node inside of the deployment XML file will now include a new node "<rmetric name='time' value='7.8988e05' unit='seconds'/>".

F. Analyzer. The name analyzer performs a hypothesis test with H0 equaling "The objective function of the modeled system will not correlate with the objective function of the profiled system." Future works on this component will examine detecting the number of iterations before the iterative improvements drops below a threshold. Additionally, future works will consider adding additional hardware metrics and trying to detect correlations between these metrics and poor solution performance, which may be useful for assisting algorithm designers in adding critical details to their models.

4.2.3 Related Work

Validation of Timing Constraints. Ha and Liu present work on validating timing constraints when jobs are independent and their execution time is only roughly known *a priori* [55]. Similar work has been done to validate timing constraints when scheduling periodic recurring tasks.

Comparison Using Algorithm Metrics [46] present a comprehensive comparison of multiple 'Directed Acyclic Graph scheduling' algorithms, using as a basis for comparison metrics such as the shortest schedule length (e.g. makespan) achieved, the number of processors used, the algorithm running time, and the range and frequency of schedule lengths. The comparison did consider that algorithms are frequently created to operate not on a real system, but on a model of multiprocessor scheduling that was similar to a real system. However, all solution metrics were calculated by 'running' scheduling plans on the model of multiprocessor scheduling instead of a real system. Therefore, if the model did not include potentially important effects, such as network contention, then metrics such as total makespan cannot be considered accurate for a real system. Our work extends this by examining the result of using algorithm solutions on a real system instead of on a system model.

4.2.4 Concluding Remarks and Lessons Learned

This work examined the challenge of validating Deployment Optimization Algorithms, or of ensuring that Deployment Optimization Algorithms actually resulted in improved performance metrics on real systems. We also introduced some of the primary challenges to the practical use of Deployment Optimization Algorithms, including the difficulty of validating the algorithms and the unknown benefits of increased optimization time. This section addresses these concerns by automating one validation procedure for Deployment Optimization Algorithms. The chosen procedure is encapsulated in DOVE, and is shown to be simple to integrate into an existing Deployment Optimization Algorithms.

4.3 BUILDING PERFORMANCE-POWER TRADEOFF MODELS OF MULTICORE SERVERS FOR DEPLOYMENT OPTIMIZATIONS

Data centers comprise substantial number of multicore server machines. In the context of these multicore server machines, power and performance tradeoffs are critical and challenging issues faced by cloud service providers (CSPs) while managing their data centers. On the one hand, CSPs strive to reduce power consumption of their data centers to not only decrease their energy costs but to also reduce adverse impact on the environment. On the other hand, CSPs must deliver performance expected by the applications hosted in their cloud in accordance with predefined Service Level Agreements (SLAs). Not doing so will lead to loss of customers and thereby major revenue losses for the CSPs. Addressing these dual set of challenges is hard for the CSPs because power management and performance assurance are conflicting objectives, particularly in the context of multi-tenant cloud systems where multiple virtual machines (VMs) may be hosted on a single physical server. The problem becomes even harder when real-time applications are hosted in these VMs. To address these challenges and make appropriate tradeoffs, we present iPlace, which is an intelligent and tunable power- and performance-aware VM placement middleware. The placement strategy is based on a two-level artificial neural network which predicts (1) CPU usage at the first level, and (2) power consumption and performance of a host machine at the second level that uses the predicted CPU usage. The efficacy of iPlace is evaluated in the context of a VM consolidation algorithm that is applied to running virtual machines and host machines in a private cloud.

4.3.1 Motivation and Problem Statement

Cloud data centers are massive-scale farms of networked multicore servers and other resource types, such as storage, that are used to host different kinds of services simultaneously from multiple different customers. Due to the use of commodity hardware for the resources, failures are common within data centers that can cause some disruptions in the hosted services. Another major factor that can cause disruptions in data centers stems from the massive power requirements of the data centers both to operate the hardware as well as for cooling. Power outages due to excess demand can result in substantial disruptions to the data center. For instance, several prominent cloud service providers (CSPs) reported days-long partial or complete outages of their cloud services platform. As an example of the adverse impact this can cause, in 2012 an intrinsic outage in Amadeus airline reservation system's data center triggered long lines and delays at many airports worldwide. Power outages are also shown to have an adverse impact on environment because they produce diesel exhaust.

Reducing energy consumption in data centers is thus an important criterion to reduce the chances of outages. This issue is particularly important considering an increasing trend towards hosting applications with soft real-time requirements in the cloud [56][57], which cannot sustain significant service disruptions. For these applications, performance concerns, such as response time and service availability, are vital requirements and hence disruptions in data centers is often not desirable.

One promising approach to maintaining availability and performance requirements of real-time applications after partial disruptions within the same data center is via live migration [58] of virtual machines (VMs). VM migrations help in hardware maintenance, fault-tolerance and

load-balancing. However, live migration may incur significant cost in terms of substantial network usage particularly when multiple simultaneous VM migrations are active at any given time thereby adding to the energy consumption. One key reason for the increased network usage is that existing approaches that use live VM migrations often tend to ignore the placement issues for the backup VMs, which in turn leads to unwanted usage of network and other resources thereby causing increased energy consumption.

For example, cloud-based high availability and performance solutions such as Remus[59], Paratus [60], and Kemari [61], require capturing the entire executions of the VM and transferring them to the backup machine as swiftly and seamlessly as possible. While these solutions are much desirable for maintaining application performance and availability, they tend to shift the responsibility of choosing the backup VMs to the cloud user. A solution that will relieve the cloud user of these responsibilities and automate the choice of backup VMs is desirable. In prior work [62][63] we addressed this limitation in the Remus high availability solution by providing a backup VM placement mechanism that was based on simple bin packing heuristics. However, this work did not consider energy conservation as a criterion.

Another technique used by CSPs to improve resource utilization, reduce energy consumption, and thereby saving on energy bills is to employ *Resource Overbooking* [64][65][66]. Even in the case of resource overbooking, the placement of VMs on aptly suited host machines where SLA is not violated is crucial. We have observed that even idle VMs that are overbooked on a host machine might affect the performance of applications running in other collocated VMs on that host because of performance interference between collocated VMs [67][68][69] when resources are overbooked. Thus, the need for effective VM placement is a key requirement.

In summary, energy conservation in data centers is increasingly becoming the focus of CSPs who are seeking ways to save on energy bills, reduce the chances of outages, and reduce adverse impact on the environment. Reducing energy consumption would imply shutting down large portions of the data center and employing resource overbooking. However, a naive approach to conserving energy may lead to applications not meeting their performance requirements, which is not acceptable to real time applications hosted in the cloud. Techniques that support both performance and availability in the cloud must continue to work. As we have seen above, a common theme that pervades these requirements is the need for effective VM placement in the data center. A common practice for VM placement decisions at the hypervisor level is bin packing heuristics such as first-fit, best-fit, and next-fit. However, these bin packing techniques do not consider power concerns of the CSPs nor performance requirements of applications.

To address these objectives, this section presents *iPlace,* which is an intelligent and tunable power- and performance-aware virtual machine placement technique that is realized as cloud infrastructure middleware. The key contributions of iPlace include:

- An intelligent tunable power- and performance-aware virtual machine placement strategy in virtualized environments that satisfies soft real-time application QoS. The novelty of our VM placement approach stems from its use of a two-stage neural network which predicts (1) CPU usage at the first level and (2) uses the predicted CPU usage at the first level to

predict the power consumption and performance of a host machine at the second level. Section 4.3 delves into the details of this contribution.

- It analyzes how energy consumption of data centers can be reduced while performance of soft real-time applications are ensured by employing iPlace. Section 4.4 presents results of our empirical studies.

The remainder of this chapter is organized as follows: Section 4.1 introduces, compares, and discusses several recent prior efforts synergistic with iPlace; Section 4.3 provides an architectural view of the iPlace middleware and the design of the two-stage artificial neural network; Section 4.4 presents the test and evaluation results of iPlace; and finally, Section 4.5 presents concluding remarks alluding to future work.

4.3.2 Related Work

This section explores prior work that employ schemes like live migration and server consolidation techniques that aim to address one or more of the performance, availability and energy consumption issues in cloud data centers.

Akiyama et. al propose MiyakoDori [70] which employs “memory reusing” technique to reduce the amount of memory transferred thereby reducing unnecessary energy consumption during the live migration. When a virtual machine monitor (VMM) initiates a live migration command, MiyakoDori retains the memory image of the VM on the source node. Identical memory pages are not transferred; only the manipulated memory content is transferred when that VM is migrated back to the original node. MiyakoDori saves substantial amount of memory between migrations thereby reducing energy consumption. This related work considers identical memory pages to reduce the energy consumption during live migrations whereas our work focuses on both reducing energy consumption and guaranteeing application performance. It is feasible that our work can leverage MiyakoDori in the live migration process.

Deshpande et al. address the problem of migrating several collocated VMs simultaneously [71]. In a data center, it is highly likely that collocated VMs might have the same operating system, similar software, and libraries installed on it. Therefore, the basic idea in this section is transferring identical contents across the collocated VMs only once. Our work is once again complementary to this approach since we focus on finding an aptly suited host machine for a VM. Thus, it is possible for our approach to leverage this related work for additional benefits.

The work closest to ours is by Hirofuchi et al. [72][73] who propose an energy-efficient VM consolidation technique for optimizing VM locations to achieve energy savings while guaranteeing performance. In this work, post-copy live migration is utilized as opposed to pre-copy live migration since post-copy migration reacts to sudden load changes more quickly than pre-copy. Data center servers are categorized as shared and dedicated servers. Shared servers host the idle VMs while dedicated servers host CPU-intensive VMs. Shared servers take advantage of extra memory to host many idle VMs. The technique utilized in their paper is to migrate as many idle machines into shared servers as possible from dedicated servers and finally switch-off the dedicated servers in which no more VMs are left. Our work also comes under the purview of consolidation algorithms. In contrast to this work, our work does not differentiate between shared and dedicated servers, which reduces the complexity of our technique.

Berral et al. [74] propose a framework that provides intelligent dynamic consolidation of VMs in which deadline-sensitive applications are executing. A machine learning-based technique is employed to reduce the energy consumption while meeting SLA requirements for high performance computing (HPC) environments where applications have deadline constraints. This work differs from our work in that their work targets a HPC environment, which are more controlled and where exclusive access to resources is granted is targeted, whereas we primarily target public cloud environments.

Piao [75] proposes a VM placement and migration approach to optimize the heavy data transfer over the network. Due to the nature of network- and data-intensive workloads, applications hosted on various VMs often communicate with each other frequently over the network which in turn adversely affects the application performance and network overhead. Moreover, it might lead to network congestion and unexpected network latency. Therefore, migrating these kinds of applications within a close proximity of their counterparts reduces the traffic on the network and ultimately optimizes the performance. The work in that paper differs from our work in that it attempts migrating highly coupled VMs to closeby locations. In contrast, we target compute-intensive applications and discover aptly suited host machine for their VMs with respect to power and performance. As future work, we will consider accounting for network usage as suggested in this prior work.

Khosravi et al. [76] have taken into account carbon footprint rate and power usage effectiveness (PUE) for designing VM placement strategy in data centers. The VM placement problem is considered as a bin packing problem with (*datacenter*×*cluster*×*host*) placement options. The authors propose Energy and Carbon-Efficient (ECE) VM placement algorithm based on best-fit heuristic to find a solution to the problem, and evaluate the algorithm using simulation. A difference with our work is that we have evaluated our solution inside a cloud data center and applied machine learning to account for a large set of factors affecting power and performance which are difficult to model in the system. In this related work, the authors considered power consumption as a function of CPU frequency, whereas we have taken several factors including memory, network, overbooking rate etc. into account for predicting performance and power.

Dong et al. [77] propose a VM placement scheme to reduce both the number of physical machines and network elements in a data center to reduce overall energy consumption. The optimization of physical servers is considered as a bin packing algorithm, while network optimization is formulated as a quadratic assignment problem. The proposed method is a combination of hierarchical clustering and best-fit to solve the optimization problem for VM placement and is evaluated based on simulations. In contrast, our work provides an intelligent placement algorithm which considers VM-based host overbooking, power consumption and performance, which is evaluated in a real data center.

4.3.3 Virtual Machine Placement using iPlace

Figure 13 depicts the strategy of iPlace, which is our intelligent power- and performance-aware virtual machine placement algorithm. The goal of iPlace is to find an aptly suited host machine by carefully considering the energy efficiency of the data center and performance requirements of soft-real time applications running on host machines. iPlace takes power changes and performance effects to the applications running on VMs for its placement decision. A tunable pa-

rameter named *performance preference level* is provided to iPlace in advance to set the performance requirement.

To find the aptly suited host machine, a two-level artificial neural network (ANN) is employed by our VM placement middleware, which are at the core of our system design and serve as the predictor mechanism. To train the ANNs, iPlace employs the Levenberg-Marquardt back-propagation algorithm [78]. At the first level, the mean CPU usage of a host machine after a VM was to be migrated to it is predicted by running the CPU usage predictor ANN. Subsequently, this predicted CPU usage value is utilized by the second level ANN. At the second level, power consumption and mean performance of the host machine is predicted by the power and performance predictor ANN. At runtime, the middleware will consult the prediction engine and if the predicted values are acceptable, the middleware will take the decision of placing the VM on a given host.

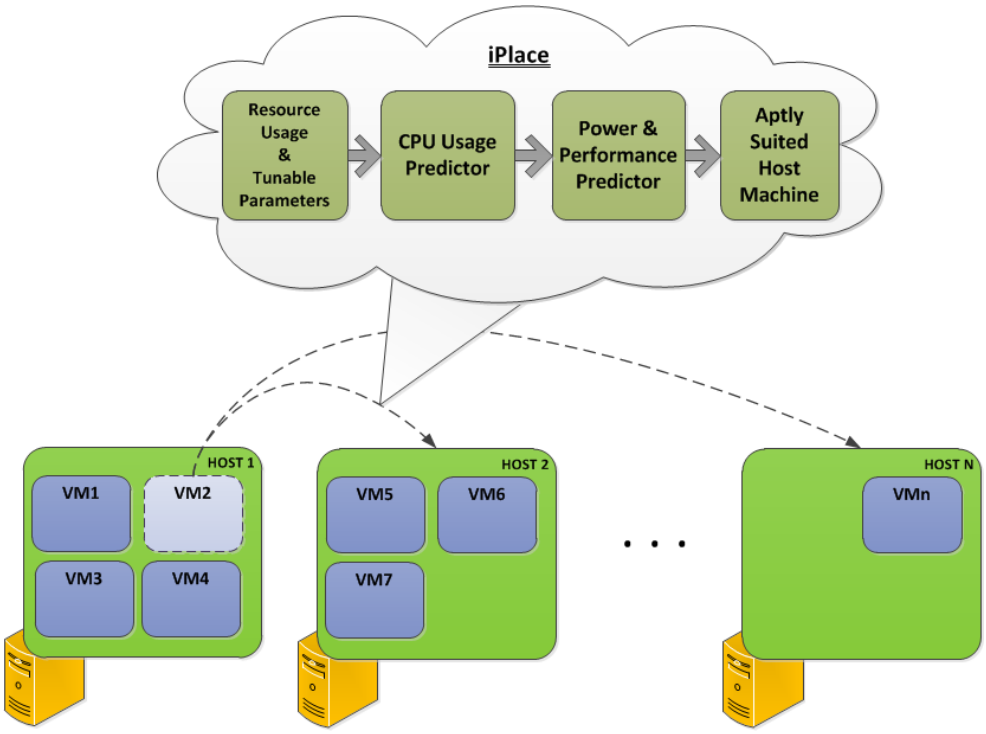


Figure 13: Illustration of iPlace's Virtual Machine Placement Strategy

To understand how these ANNs are used to make runtime decisions, consider the case when one of the consolidation algorithms, high availability solutions, or scheduling mechanisms would like to migrate a VM from one host machine to another one. iPlace finds the aptly suited host machine by predicting the power consumption and performance values for each host machines in the cluster as though the VM was migrated on to it. As illustrated in Figure 13, iPlace employs both CPU usage predictor and power and performance predictor sequentially by feeding their required input values.

In our current design, iPlace targets only compute-intensive applications, therefore 1/(*CPUtime*) metric was utilized in this work as the performance indicator of an application.

The higher the performance value, the better the performance. Additionally, we assume that CSPs overbook their underlying cloud infrastructure to save energy costs. Details of the ANNs are described below.

4.3.3.1 CPU Usage Predictor

The structure of the CPU usage predictor ANN is depicted in Figure 14. The purpose of the CPU usage predictor is to estimate the amount of CPU usage of the host machine after a VM were to migrate onto it. Due to the CPU contention in over utilized virtualized environments, the mean CPU usage of a host machine might not increase by the same amount of CPU usage currently been illustrated by the VM being migrated. Thus, a simple subtraction on one machine and addition on another machine does not work. Therefore, iPlace employs a CPU usage predictor ANN for this prediction so that the power consumption and performance of the host machine could be determined effectively by knowing the CPU usage of the host machine.

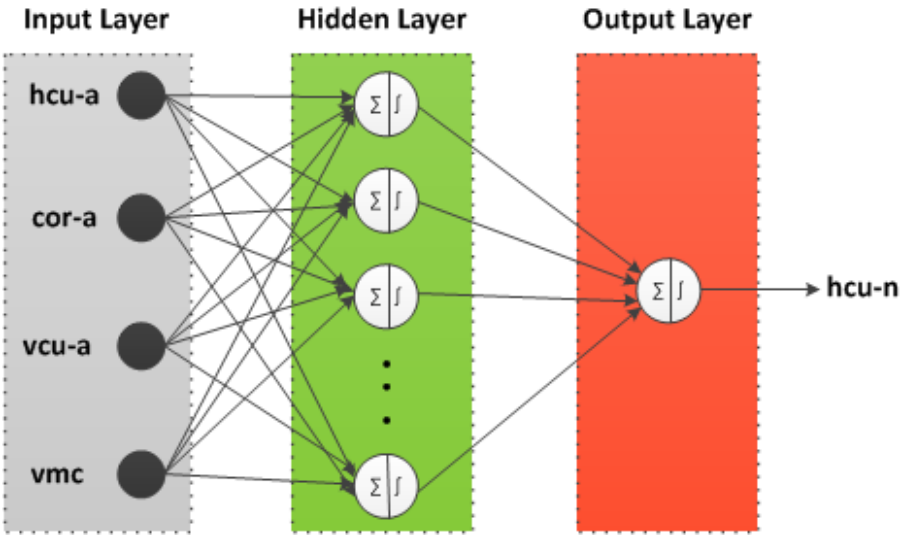


Figure 14: Structure of the CPU Usage Predictor ANN

The topology of the CPU usage predictor ANN is shown below.

InputLayer	:*hcu−a,cor−a,vcu−a,vmc*
HiddenLayer	:9 *neurons*
ActivationFunction	(*in hidden layer*)
	:*Tangent Sigmoid*
OutputLayer	:*hcu−n*
TransferFunction	(*in output layer*)
	:*Pure Linear*
where	
hcu−a	*= Actual mean CPU usage of the host machine before VM is migrated on it*
cor−a	*= Actual CPU overbooking ratio of the host machine before VM is migrated on it*
vcu−a	*= Actual CPU usage of the VM being*

migrated onto the host machine

vmc = *Actual VM count on the host machine before VM is migrated on it*

hcu−n = *CPUusage of the host machine after VM is migrated onto it*

The CPU overbooking ratio and mean CPU usage of the host machine provided to this ANN are computed by Equations (11) and (13).

$$Total\ vCPU\ Requested = \sum_{i=0}^{m} vCPU_i \tag{10}$$

$$CPU\ Overbooking\ Ratio = \frac{Total\ vCPU\ Requested}{Total\ pCPU\ Cores} \tag{11}$$

$$Total\ CPU\ Usage = \sum_{i=0}^{m} vmCPUUsage_i \tag{12}$$

$$Host\ Mean\ CPU\ Usage = \frac{Total\ CPU\ Usage}{m} \tag{13}$$

where

Total vCPU Requested: *Totalnumberofvirtual CPU coresrequestedonahost machine*

m: *Totalnumberoftheguest VMsonahostmachine*

vCPU: *NumberofvirtualCPUcores ofaVM*

Total pCPU Cores: *Totalnumberofphysical CPUcoresofahostmachine*

The number of neurons in the hidden layer is determined based on experimentation by trying different numbers and examining the system results. The reliability and accuracy of ANNs employed by iPlace is examined by carefully looking into the mean squared error (MSE) and regression (R) values. The MSE value provides average squared difference between input and output whereas the R value describes how the input of the system is correlated with its output. The best performance of the CPU usage predictor ANN was produced with 9 neurons in the hidden layer, and MSE of 0.00044 and R of 0.99. As shown in Figure 15, these MSE and R values clearly indicate that CPU usage predictor precisely estimates the host machine's CPU usage.

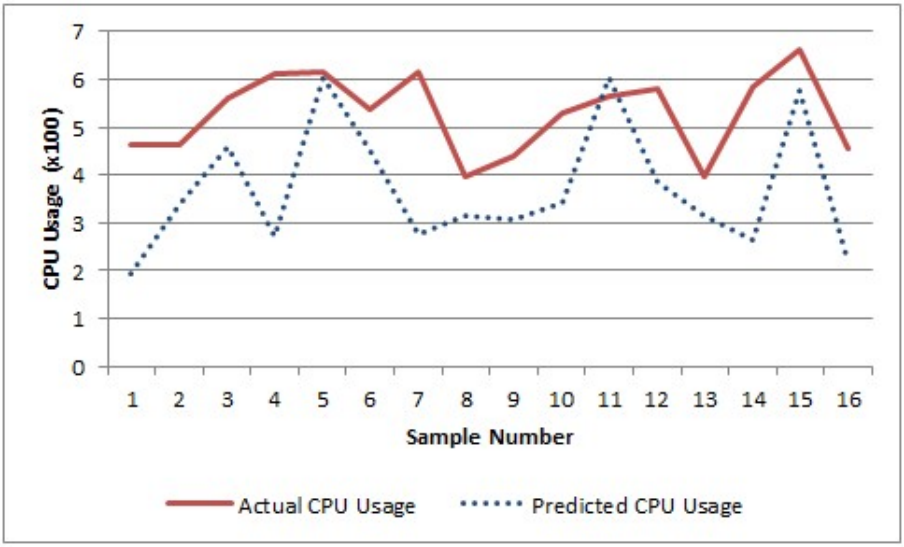


Figure 15: Comparison of Actual and Predicted CPU Usage of Host Machine Power and Performance Predictor

4.3.3.2 Power/Performance Predictor

The structure of the power and performance predictor ANN is depicted in Figure 16. The output of the CPU usage predictor ANN is provided as input to the power and performance predictor ANN. The purpose of power and performance predictor is to predict power consumption and performance of the host machine if the VM were to migrate to it.

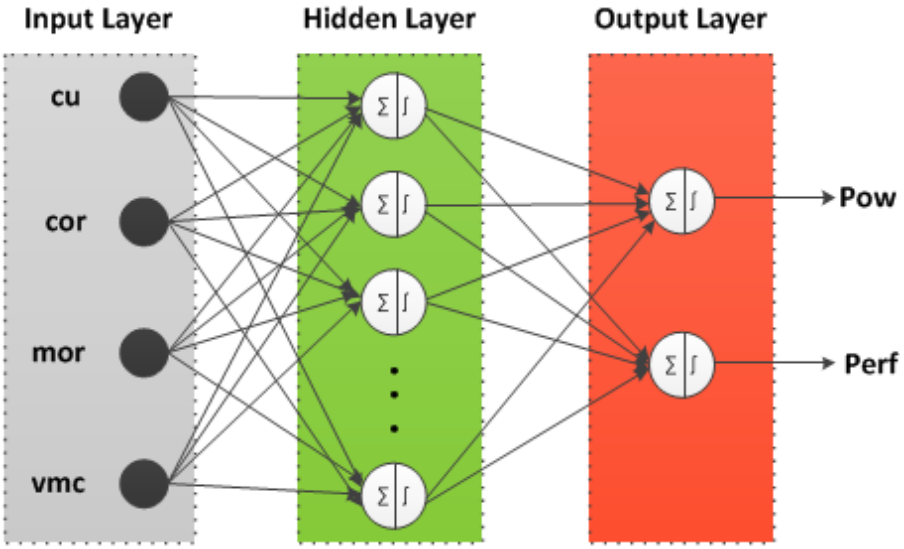
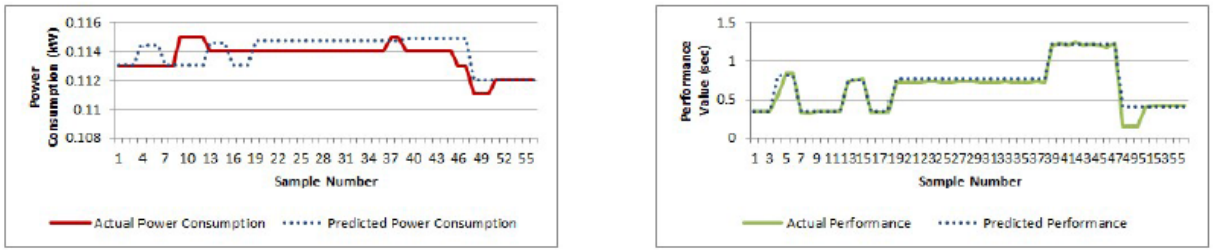


Figure 16: Structure of the Power and Performance Predictor Artificial Neural Network



(a) Power Consumption (b) Performance

Figure 17: Comparison of Actual and Predicted Power Consumption and Performance Value Results of Host Machine

The topology of power and performance predictor ANN is detailed below.

InputLayer	:*cu, cor, mor, vmc*
HiddenLayer	:12 *neurons*
ActivationFunction	(*in hidden layer*)
	:*Tangent Sigmoid*
OutputLayer	:*Power, Perf*
TransferFunction	(*in output layer*)
	:*Pure Linear*
where	
cu	= *Mean CPU usage of the host machine*
cor	= *CPUoverbooking ratio of the host machine*
mor	= *Memory overbooking ratio of the host machine*
vmc	= *VM count on the host machine*
Pow	= *Power consumption of the host machine*
Perf	= *Performance value of the host machine* (*i.e.mean performance of all the guest VMs running on the host*) *machine*)

The best performance of the power and performance predictor ANN was produced with 12 neurons in the hidden layer, MSE of 0.008, and R of 0.97. These MSE and R values clearly show that the power and performance predictor ANN precisely estimates the host machine's power consumption and performance. Figure 17 depicts the comparison of actual power consumption and performance values of host machine along with the predicted values of power and performance predictor.

By carefully observing the data generated by our simulation software, we determined the mean performance value (μ) as 1.75 and the standard deviation (σ) as 1.17. These values are the assumed indicators for performance requirement of the soft real-time application and utilized to check whether the performance requirement of the soft real-time application validated by the Equation (14). This performance indicator is assured on the host machine where it will be mi-

grated with best effort. α in Equation(14) is basically the performance preference level parameter passed by the system user. The tighter performance requirement might cause iPlace not to be able to find any host machine.

$$Pr = \mu + \alpha * \sigma \quad (14)$$

where

Pr: *Performance requirement of the VM*

μ: *Mean performance value computed by looking the values in the cluster*

α: *Performance preference level*

σ: *Standard deviation value values in the cluster computed by looking the values in the cluster*

This performance parameter along with the resource usage information is provided to iPlace. iPlace employs the CPU usage predictor ANN first and feeds the predicted CPU usage of the host machine to the power and performance predictor ANN then. After receiving the CPU usage, iPlace finds all the host machines satisfying the performance requirement in Equation (14) by comparing these predicted performance values with the value returned from Equation (14) by starting from the *performance preference level*. If none of the host machines satisfies the requested *performance preference level*, iPlace gradually lower the *performance preference level* by one and checks each host machine again to find another host machine that will satisfy this new performance requirement. Then, iPlace computes the power change on the each host machine satisfying the performance requirement to see how much power will increase. Finally, the placement decision is made onto the host machine which satisfies the performance requirement and has the minimum power change.

4.3.4 Validating the iPlace Approach

The iPlace framework consists of two stage neural network. For accurate predictions, the training data set for both the stages should be as close as possible to real-world data. To achieve the same, we have used a private data center consisting of five hosts on which we have emulated a workload that is similar to that of a production data released by Google Inc. from one of their cluster's trace log [79]. Our private data center comprises a homogeneous set of machines and is managed by the OpenNebula cloud management solution version 3.2.1. Table 2 provides the configuration of each host used as a cluster node. Each host is connected to a *Watts Up? Pro* power meter, which can report consumed power with frequency of once a second and an accuracy of a tenth of a watt.

Table 2: Hardware and Software Specification of Cluster Nododes

Processor	2.1 GHz Opteron
Number of CPU cores	12
Memory	32 GB
Hard disk	8 TB

Operating System	Ubuntu 10.04 64-bit
Hypervisor	Xen 4.1.2
Guest virtualization mode	Para

The Google cluster trace contains a dataset for about 12,000 distinct machines collected over a 29 day period in the month of May, 2011. We chose one of the host with ID 257408495 from the cluster and reproduced the workload on one of the host in our data center for one day. The VM configuration and resource usage in the dataset was normalized which we scaled according to host configuration and pruned the data which did not fit the characteristics of our host.

The workload on the host was generated using our simulation software coded in Python which used OpenNebula to create and delete virtual machines. We executed *Lookbusy* (a synthetic load generator) processes to mimic the CPU and memory workloads. Our resource monitoring application was coded in C++, which runs to collect the resource usage information of the host using *libvirt* library at certain specified interval.

Matlab software is used to train and run the ANNs as well as deciding the placement decision. Additionally, a SQL server database management system is utilized to import the resource information data of each host machine and prepare the training set for ANNs.

Below we show the experimental results of iPlace that we have tested and evaluated by following two test cases we have defined. The initial configuration of our cluster is depicted in Figure 18. In Figure 18, the tuple under each VM name represents the resource capacity of the VM in the format of <*cpu, memory*>. Additionally, initial resource usage and overbooking ratios of each host machine in the cluster is illustrated in Table 3

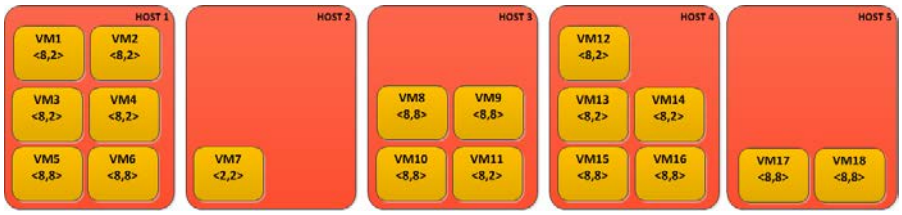


Figure 18 Initial Configuration of the Cluster Utilized in Test Cases

Table 3: Initial Resource Usage of Host Machines in the Cluster

	CPU Usage	CPU Overbooking Ratio	Memory Overbooking Ratio
HOST 1	16%	4	0.75
HOST 2	18%	0.17	0.06
HOST 3	30%	2.6	0.81
HOST 4	23%	3.3	0.69
HOST 5	1.5%	1.3	0.5

Use Case 1: In this use case, we assumed that there was an abnormal activity causing performance degradation of VM2, which is a high priority VM on Host 1. Therefore, a decision was made to migrate it to another host machine in the cluster. We analyzed and compared the results of placement decision of iPlace with a first-fit heuristic of bin packing algorithm in the context of power and performance. We have tested iPlace with three different performance preference levels (i.e. α values of -1, 0, 1).

Recall that performance preference level is the tunable performance parameter passed by the system user. It is also used in the performance requirement equation (14) with α as the parame-ter. Based on that performance preference level, the standard deviation is adjusted and perfor-mance requirement is either tightened or softened. As the performance preference level goes below 0, -1, -2 ..., the performance requirement of the application is softened and vice versa.
As expected, the placement decision of first-fit heuristic is to place the VM on the first host machine in which it fits. Therefore, a first-fit heuristic placed VM2 on to Host 2. iPlace decides the target host machine by observing power changes on host machines and performance effects on the applications running on the VM. Thus, the placement decision for each performance preference level presented in Table 4 for this use case might be dissimilar for any other use cases.

Table 4: Test Results of Use Case 1

Performance Preference Level (α)	Placement Decision
-1	HOST 5
0	HOST 5
1	NONE

As shown in Table 4, iPlace decided to migrate VM2 to Host 5 at both performance preference levels of α = -1 and α= 0. iPlace assured the performance requirement of VM2 on none of the host machines in the cluster for the tighter performance preference level of α = 1.

To detail the case where performance preference level of α =0, iPlace predicted that only Host 3 and Host 5 satisfied the performance requirement in Equation (14). However, iPlace decided to migrate VM2 onto the Host 5 due to the prediction of lower amount of power increase by 0.0274kW on Host 5 versus 0.0281kW on Host 3.

Compared to the first-fit heuristic, iPlace could not assure the performance requirement of VM2 on Host 2 even though VM2 fits on it. Therefore, it discarded Host 2 for its placement decision for VM2.

Use Case 2: In this use case, we assumed that a decision was made to migrate all VMs residing on one of the less utilized host machines onto the rest of the host machines decided by iPlace. Host 2 was selected as the target due to having only one VM. Therefore, VM7 will be migrated and Host 2 will be shut down.

At initial run with performance preference level of $\alpha = 0$, iPlace could not find a host machine for that criteria. It determined Host 3, Host 4, and Host 5 after iterating till the performance preference level of $\alpha = -2$. However, iPlace decided to migrate VM7 onto the Host 3 due to the power change concerns. These results are depicted in Table 5.

After migrating VM7 onto the Host 3, Host 2 becomes idle and started to consume 0.091kW power with no VMs running on it. Therefore, we assumed it was turned off and computed the overall power consumption of the cluster. The total power consumption of the cluster dropped from 0.708kW to 0.614kW which saved about 13% of power consumed by the cluster.

Table 5: Test Results of Use Case 2

VM Name	Performance Preference Level (α)	Placement Decision
VM17	0	HOST 3
VM18	0	HOST y

4.3.5 Concluding Remarks and Lessons Learned

In this section we described iPlace, which is an intelligent tunable power- and performance-aware virtual machine placement strategy. The work was motivated by the need to conserve energy in data centers yet manage the performance and availability requirements of soft real-time applications that are increasingly being hosted in cloud data centers. To that end, we have developed a two-level artificial neural network (ANN) with stage one responsible for CPU usage prediction, and stage two responsible for power and performance prediction. The two stage ANN was designed, trained and employed to forecast the host machine's CPU usage, power consumption, and performance. For training purposes and evaluation, we generated workloads in our private cloud that emulated data from a Google's production server. We have tested and evaluated iPlace in our private cloud and compared results with first-fit bin-packing heuristic. The results show that iPlace could help to save certain degrees of power consumption by satisfying variety of performance requirements. Compared to the first-fit heuristic, iPlace places VMs on host machines where application performance is assured and energy efficiency is maximized.

Since the private cloud environment where we tested and evaluated iPlace is a homogeneous environment, our test results were validated only in a homogeneous environment. However, iPlace could easily be employed in a heterogeneous environment by providing additional host machine capacity information parameters to the ANNs, such as number of cores and memory size. In this work, we targeted only the compute-intensive applications due to the performance metric we utilized. By integrating more generic application performance metrics, such as response time or throughput, iPlace could support a variety of application types in the cloud environment. These dimensions will form the basis of our future work.

4.4 ABSTRACT EMULATION OF CPU WORKLOAD GENERATION

Enterprise distributed real-time and embedded (DRE) systems are increasingly using high-performance computing architectures, such as dual-core architectures, multi-core architectures, and parallel computing architectures, to achieve optimal performance. Performing system integration tests on such architectures in realistic operating environments during early phases of the software lifecycle, *i.e.*, before complete system integration time, is becoming more critical. This helps distributed system developers and testers evaluate and locate potential performance bottlenecks before they become too costly to locate and rectify. Traditional approaches either (1) rely heavily on simulation techniques or (2) are too low-level and fall outside the domain knowledge distributed system developers and testers. Consequently, it is hard for distributed system developers and testers to produce realistic operating conditions for early integration testing of such systems.

This section provides two contributions to facilitating early system integration testing of enterprise DRE systems. First, it provides a generalized technique for emulating computation intensive workload irrespective of the target architecture. Secondly, this article illustrates how the emulation technique is used to evaluating different high-performance computing architectures in early phases of the software lifecycle. The technique presented in this article is empirically and quantitatively evaluated in the context of a representative enterprise DRE system from the domain of shipboard computing environments.

4.4.1 Emerging trends in enterprise distributed real-time and embedded systems.

Enterprise distributed real-time and embedded (DRE) systems, such as mission avionics systems, shipboard computing environments, and traffic management systems, are increasingly using high-performance computing architectures [91, 96], *e.g.*, multi-threaded, hyper-threaded, and multi-core processors. High-performance computing architectures help increase parallelization of computation intensive workload, which in turn can improve overall performance of enterprise DRE systems [96]. Furthermore, such computing architectures enable enterprise DRE systems to scale to the computation needs of next generation enterprise DRE systems, such as ultra-large-scale systems [92].

As enterprise DRE systems grow in both complexity and scale, it is becoming more critical to evaluate the system under development on different high-performance computing architectures early in the software lifecycle, *i.e.*, before complete system integration time. This enables distributed system developers to determine which architecture is best for their needs. It also helps distributed system developers avoid the *serialized-phasing development problem* [98] where infrastructure- and application-level system entities are developed and validated in different phases of the software lifecycle, but fail to meet performance requirements when integrated and deployed together on the target architecture. Serialized-phasing development therefore makes is hard for distributed system developers and testers to identify potential performance bottlenecks before they become too costly to locate and rectify.

Existing techniques for evaluating and validating computation intensive systems on high-performance computing architectures early in the software lifecycle to overcome the serialized-phasing development problem rely heavily on simulation and/or analytical models [81-85]. Although this approach is feasible for predicting performance in early phases of the software lifecyle [97], such techniques cannot account for all the complexities of enterprise DRE systems (*e.g.*, the operating environment and underlying middleware. These complexities and others, such as arrival rates of events and thread/lock synchronization, are known to affect computation intensive workload and overall performance of such systems. Distributed system developers therefore need improved techniques for evaluating enterprise DRE systems on different high-performance computing architectures in early phases of the software lifecycle.

Solution approach. Abstracting computation intensive workload via emulation techniques. Emulation [97, 99] is an approach for constructing operational environments that are capable of generating realistic results. In the context of high-performance computing architectures, emulating computation intensive workload on the target architecture and operational environment enables distributed system developers and testers to construct realistic operational scenarios for evaluating different high-performance computing architectures. Moreover, it allows distributed system developers and testers to evaluate and validate system performance, such as latency, response time, and service time, when complexities of enterprise DRE systems are taken into account, such as the underlying middleware, target architecture, and operating environment.

Below we describe a technique for emulating computation intensive workload to evaluate different high-performance computing architectures and evaluate and validate enterprise DRE system performance early in phases of the software lifecycle. The emulation technique presented in this article abstracts away optimizations of high-performance computing architectures, such as caching and look-ahead processing, while letting the target architecture handle execution concerns, such as context-switching and parallel processing, which can vary between different high-performance architectures. The emulation technique is able to accurately emulate computation intensive workloads ranging from [1, 1000] msec irrespective of the underlying high-performance computing architecture. Experience gained from applying the emulation technique presented in this article shows that it raises the level of abstraction for performance testing so that distributed system developers and testers focus more on locating potential performance bottlenecks instead of wrestling with low-level architectural concerns when constructing realistic operational environments.

4.4.2 Case Study: The SLICE Scenario

The SLICE scenario is a representative enterprise DRE system from the domain of shipboard computing environments. It has be used in prior research and multiple case studies, such as evaluating system execution modeling tools [87, 90], conducting formal verification [88], and highlighting challenges of searching the deployment and configuration solution space of such systems [89]. Figure 19 shows a high-level model of the SLICE scenario.

To briefly reiterate an overview of the SLICE scenario, Figure 19 illustrates that the SLICE scenario is composed of seven different component instances named: (from left to right): *SenMain*, *SenSec*, *PlanOne*, *PlanTwo*, *Config*, *EffMain*, and *EffSec*. The directed lines between

each component represents inter-communication between components, *e.g.*, sending/receiving an event. The SLICE scenario also has a *critical path* of execution, which is represented by the solid directed lines between components that must be executed in a timely manner. Finally, each of the components in the SLICE scenario must be deployed (or placed) on one of three hosts in the target environment.

The SLICE scenario is an application-level entity, however, the infrastructure-level entities it will leverage are currently under development. The SLICE scenario is therefore affected by the serialized-phasing development problem described earlier. To overcome the effects of serialized-phasing development, system developers are using system execution modeling tools to conduct system integration test for the SLICE scenario, such as evaluating end-to-end response time of its critical path, during early phases of the software lifecycle, *i.e.*, before complete system integration.

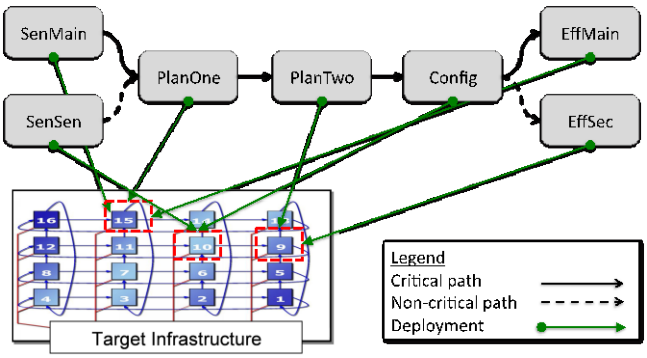


Figure 19 High-level overview of the SLICE scenario.

In particular, system developers are using the Component Utilization Test Suite (CUTS) [90] system execution modeling tool, which is designed for component-based distributed systems, to conduct system integration tests during early phases of the software lifecycle, and to overcome the serialized-phasing development problem. CUTS uses domain-specific modeling languages [95] and emulation techniques to enable system integration testing during early phases of the software lifecycle on the target architecture using components that look and feel like their counterparts under development. Likewise, as the *real* components are developed, they can seamlessly replace the *faux* components to facilitate continuous system integration testing throughout the software lifecycle.

Similar to prior work [90], system developers intended to use CUTS to evaluate the end-to-end response time of the SLICE scenario's critical path. This time, however, system developers plan to extend their previous testing efforts and evaluate the end-to-end response time of the SLICE scenario's critical path on different high-performance computing architectures, such as multi-threaded, hyper-threaded, and multi-core architectures. Applying CUTS to evaluate the SLICE scenario on different high-performance computing architectures, however, revealed the following limitations:

- **Limitation 1. Inability to easily and accurately adapt to different high-performance**

computing architectures. Different high-performance computing architectures have different hardware specifications, such as processor speed and number of processors. Different hardware specifications also affect the behavior of computation intensive workload. For example, execution time for two separate threads on a dual-core architecture will be less than execution time for the same threads on a single processor architecture—assuming there are no blocking affects, such as waiting for a lock, between the two separate threads of execution.

System developers can use the high-performance computing architecture's hardware specification to model the behavior of computation intensive workload for emulation purposes. This approach, however, is too low-level and outside their knowledge domain. Likewise, such a model can be too costly to construct and validate, and can negatively impact overall emulation performance on the target architecture. System developers therefore need a technique that can easily adapt to different high-performance computing architecture and provide accurate emulation of CPU workload.

- **Limitation 2. Inability to adapt and scale to large number of computation resources.** System developers plan to leverage dynamic testbeds, such as Emulab [100], to conduct their integration tests. Emulab enables system developers to configure different topologies and operating systems to produce a realistic target environment for distributed system integration testing.

Although the SLICE scenario consists of three separate hosts, system developers intend to conduct scalability tests by increasing both the number of components and hosts in the SLICE scenario. System developers therefore need lightweight techniques that will enable them to rapidly include additional resources in their experiments, and still provide accurate emulation of computation intensive workload.

Due to these limitations it is hard for system developers of the SLICE scenario to evaluate the end-to-end response time of its critical path on different high-performance computing architecture. Moreover, this problem extends beyond the SLICE scenario and applies to other distributed systems that need to evaluate system performance on different (high-performance) computing architectures. The remainder of this article, therefore, discusses a technique for overcoming the aforementioned limitations and improving CUTS to enable early system integration testing of computation intensive enterprise DRE systems on different high-performance computing architectures.

4.4.3 Architecture Independent Approach for Accurate Emulation of CPU Workload

Below we present a technique for accurately emulating computation intensive workload independent of the underlying high-performance computing architecture. The approach abstracts CPU workload and focuses on overall execution time of the computation intensive workload under emulation.

4.4.3.1 Abstracting CPU Workload for Emulation

Conducting system integration test (1) during early stages of the software lifecycle, (2) on the target architecture, and (3) in the target environment enables distributed system developers to obtain realistic feedback for the system under development. Moreover, it enables distributed system developers to locate potential performance bottlenecks so they can be rectified in a timely manner. The ability to locate such performance bottlenecks, however, depends heavily on the accuracy of such tests, *i.e.*, its behavior and workload, conducted during early phases of the software lifecycle.

In the context of high-performance computing architectures, it is possible to construct fine-grained models that will accurately emulate effects of their characteristics, such as enhanced performance due to CPU caching or look-ahead processing, or fetching instructions from memory. For example, Figure 20 illustrates an emulation/execution model for (a) fetching instructions from memory and (2) CPU caching effects.

As illustrated in Figure 20, in the case of (a) fetching instructions from memory, a portion of the computation intensive workload (*i.e.*, overall execution time) is attributed to fetching the instructions from memory. Likewise, in the case of (b) CPU caching effects, the portion of the overall execution time that would have been attributed to fetching CPU instructions from memory no longer exists.

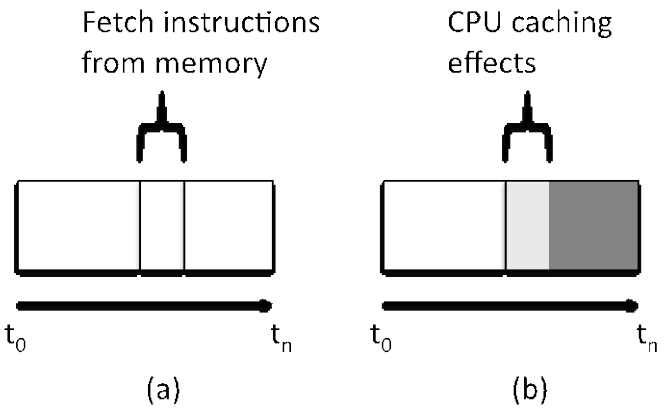


Figure 20 Emulation/execution model to high-performance computing effects.

Because high-performance computing architectures have many characteristics that can impact performance, it is not feasible to construct fine-grained emulation models that capture all effects—especially when conducting system integration tests during early phases of the software lifecycle. Instead, a more feasible approach is abstracting away such effects and focusing primarily on overall execution time, similar to profiling [86]. This is possible because as the same computation (re)occurs many times throughout the lifetime of a system, it will converge on an average execution time. The average execution time will therefore incorporate the effects from characteristics of its underlying high-performance computing architecture.

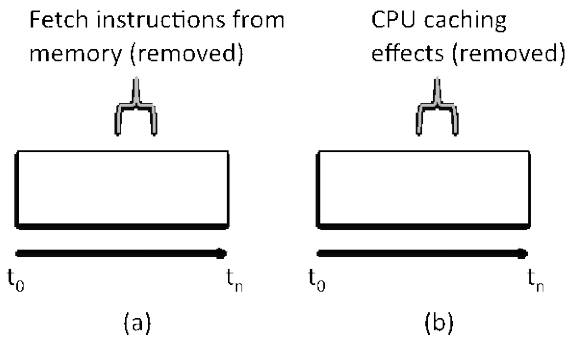


Figure 21 Example of abstracting the emulation/execution model for computing effects.

The figure above highlights abstracting the effects presented in Figure 20. As shown in Figure 21, instead of modeling each individual effect, the average execution time for the computation intensive workload is modeled. For example, (a) does not explicitly model the effects of fetching instructions from memory. Instead (a) models the average execution time of the computation intensive workload, which includes the execution time for occasionally fetching instructions from memory. The same holds to be true for (b), *i.e.*, instead of modeling performance gain from CPU caching effects, the average execution time for the computation intensive workload is modeled, which includes such effects. The remainder of this section discusses how abstracting CPU effects is realized when emulating computation intensive workload for enterprise DRE systems.

4.4.3.2 Realizing Abstraction of Computation Intensive Workload in CUTS

In Section 4.4.3.1, a technique for abstracting the different CPU effects of high-performance computing architectures was discussed. The main thrust of the technique focuses on overall average execution time of a computation intensive workload (or emulated CPU workload), instead of modeling each individual CPU effect—especially when conducting system integration tests during early phases of the software lifecycle. This, in turn, simplifies emulating computation intensive workload for different high-performance computing architectures.

Since overall average execution time is the main focus of the abstraction technique, it is therefore feasible to use an arbitrary computation to represent emulated CPU workload. The main challenge, however, is ensuring the arbitrary computation is capable of accurately representing different average execution times, *i.e.*, scaling to different CPU (or computation) workloads. In CUTS (see Section 5.2), this challenge is resolved by using a calibration factor for an arbitrary CPU workload. The calibration factor is then used to scale the arbitrary CPU workload to different computational needs, *i.e.*, different average execution times. Algorithm 1 presents the algorithm for calibrating the CPU workload generator that realizes the abstraction technique in CUTS.

Algorithm 1 General algorithm for calibrating CPU workload generator that abstracts CPU effects.

```
1:  procedure CALIBRATE(f, δ, max)
2:      f: target scaling factor for workload
3:      δ: acceptable error in calibration
4:      max: maximum number of attempts for calibration
5:      i ← 0
6:      bounds ← {0, MAX_INT}
7:      calib ← 0
8:
9:      while i < max do
10:         while bounds[0] ≠ bounds[1] do
11:             calib ← (bounds[0] + bounds[1])/2
12:             t ← exec(computation, calib)
13:
14:             if t > f + δ then
15:                 bounds[1] ← calib
16:             else if t < f − δ then
17:                 bounds[0] ← calib
18:             else
19:                 break
20:             end if
21:         end while
22:
23:         done ← VERIFY(calib)
24:         if done = true then
25:             return calib
26:         end if
27:     end while
28: end procedure
```

As illustrated in Algorithm 1, the goal of the calibration effort is to derive a calibration factor *calib* that will achieve the scaling factor *f* (or time). As highlighted in Algorithm 1, the initial bounds of the calibration factor is defined as for each iteration at deriving the correct calibration factor for the CPU workload (or abstract computation), the median value is used as the potential calibration factor. If the calibration factor yields an execution time above the target scaling factor, then the current calibration factor becomes the upper bound. Likewise, if the calibration factor yields an execution time below the target scaling factor, then the current calibration factor becomes the lower bound.

This process continues until either (1) the calibration factor yields an execution time that is within acceptable error of the target scaling factor or (2) the lower and upper bounds are equal. In the case that the lower and upper bounds are equal and the calibration factor is not derived, the calibration effort is tried for up to *max* times. This can occur if there is background noise during the calibration exercise. If a calibration factor is derived, then it is verified that it will accurately generate execution times up to a user-defined execution time by scaling the calibration factor based on the number of iterations need to reach a target execution time. Finally, if the verification process fails, then the average error of the verification process for different exe-

cution times is used to adjust the current calibration factor for the next attempt.

By using the calibration exercise presented in Algorithm 1 it is possible to accurately emulate CPU workload independent of the underlying computational resources on different high-performance computing architectures, which has been realized in CUTS. Moreover, it enables emulation of computation intensive workload based on CPU time and (1) not "wall time" or (2) having to monitor how much time a given thread has currently executed on the CPU.

4.4.4 Evaluating Abstraction Technique for Emulating Computation Intensive Workload

Below we present the results for validating the computation intensive workload generator discussed in Section 5.3. This section also presents the results for applying the computation intensive workload generator to the SLICE scenario introduced in Section 5.2. It is necessary to validate the workload generator because it will ensure that system developers of the SLICE scenario are able to accurately emulate CPU workload for their early integration tests. Moreover, as components are collocated (*i.e.*, placed on the same host), the average execution time of its computation intensive workload is expected to be longer due to software/hardware contention [17, 99] than when the same components are deployed in isolation. It is therefore necessary to validate that the abstraction technique can produce such behavior/results.

The experiments described below (unless mentioned otherwise) were run in a representative target environment testbed at the System Integration (SI) Lab @ IUPUI (www.emulab.cs.vanderbilt.edu). SI Lab is powered by Emulab software that configures network topologies and operating systems to produce a realistic target environment for distributed system integration testing.

4.4.4.1 Validating the Calibration and Emulation Technique

Determining the upper bound of the emulation technique discussed in Section 5.3 will enable distributed system developers to understand how much CPU workload can accurately be guaranteed before the emulation becomes unstable. When calibrating the CPU workload generator using Algorithm 1 in Section 5.3 to determine this upper bound, all tests were conducted in an isolated environment on the target host. This prevents any interference from other process that many be executing in parallel on the target host. Figure 23 highlights results of the calibration exercise when the scaling factor *f* is 20000 usec, acceptable error δ is 100 usec, and *max* is 10.

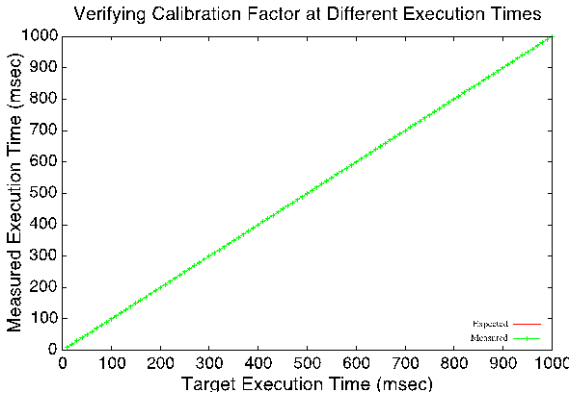


Figure 22 Calibration results for the CUTS CPU workload generator.

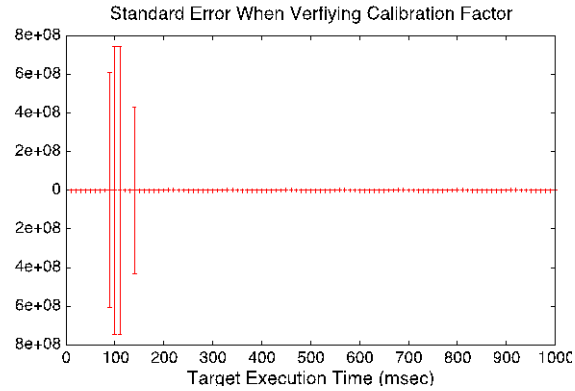


Figure 23 Calibration results for the CUTS CPU workload generator.

As illustrated in Figure 23, once the CPU workload generator is calibrated using Algorithm 1, the emulation technique is able to accurately generate computation intensive workloads up to 1000 msec (or 1 second) as illustrated in the upper graph. The lower graph illustrates the standard error for each of the execution times during verification process where the most error occurred between [70, 150] msec. After 1000 msec of computation, the emulation becomes unstable is not able to accurately emulate the computation intensive workload (not shown in Figure 23). It is believed that the inaccuracy after 1000 msec is attributed to the fact that is hard to guarantee long running processes will occupy the CPU without real-time scheduling capabilities.

The calibration and verification results in presented in Figure 23 took only 1 try. This means that distributed system developers that who want to evaluate system performance have to ensure their CPU workload is less than 1000 msec, which is well above the upper bound for many short running computations of enterprise DRE systems, such as in the SLICE scenario. More importantly, they have a simple technique that will accurately emulate computation intensive workload without modeling low-level architecture concerns (*i.e.*, addresses Limitation 1 in Section 5.2).

Validating the calibration technique on multiple homogeneous hosts. The emulation technique presented in Section 5.3 and validated above enables system developers to accurately emulate computation intensive workload up to 1000 msec. This emulation technique, however, works only on the host on which the calibration was performed. Distributed system developers intend to use SI Lab to conduct early system integration tests of the SLICE scenario. Executing calibration test of each of the hosts in SI Lab, however, is hard and time-consuming because it requires isolated access to each host in a resource-sharing environment.

Distributed system developers of the SLICE scenario therefore hypothesize that it is possible to use the same calibration for different homogeneous hosts, or architectures. If this hypothesis is true, then it will reduce the complexity of managing and configuring integration testbeds. For example, if distributed system developers elect to use integration testbeds like SI Lab, then they have to only calibrate the CPU workload generator once on each class of hosts.

Figure 24 illustrates the results for calibrating the CPU workload generator on 77 different hosts in Emulab (www.emulab.com). Emulab was used for this experiment because it has more hosts for testing than SI Lab. Each host in Emulab used for this experiment was a Dell PowerEdge 2850s with a single 3 GHz processor, 2 GB of RAM, 2 x 10,000 RPM 146 GB SCSI disks configured with a Linux 2.6.19 Uniprocessor kernel. As illustrated in Figure 24, each host in the experiment yielded the same calibration factor. The distributed system developer's hypothesis was therefore true (*i.e.*, addresses Limitation 2 in Section 5.2). This also implied that different host with the same architecture and configuration can use the same calibration and consistently generate the same accurate CPU workload for different average execution times ranging between [1, 1000] msec.

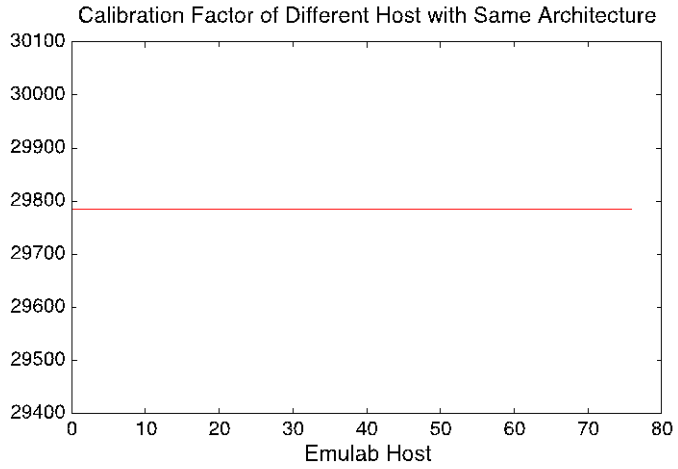


Figure 24 Validating calibration technique on different hosts with same architecture.

Validating the emulation technique against multiple threads of execution. Distributed system developers understand that as components of the SLICE scenario are collocated, the execution time (or service time) for handling events will increase. This is due to having separate

threads of execution for processing each event and hardware/software contention on the host. Distributed system developers of the SLICE scenario expect to experience similar behavior when using the CPU workload generator for early integration testing. They therefore hypothesize that the emulation technique realized in CUTS's CPU workload generator produces execution times greater than the expected execution time depending on the number of threads executing on the host.

Figure 25 presents the results for emulation 3 different threads of execution to single processor. As highlighted in Figure 25 each thread has an expected execution time: 30 msec (top), 70 msec (middle), and 120 msec (bottom). Figure 25 illustrates, however, that the measured execution time is greater than the specified execution time. This is because the CUTS's CPU workload generator is emulating CPU time instead of wall time. Moreover, when software performance engineering [97] techniques are taken into account, the measured execution time for the emulation is bounded by Equation 15,

$$S_i < D_i < n \times S_i \tag{15}$$

where is the measured service time (or service demand), *n* is the number of threads executing on the host, and is the expected execution time (or service time) of the thread.

Because of the results presented in Figure 25, distributed system developer's hypothesis about the behavior of CUTS's CPU workload generator for multiple threads of execution was correct. More importantly, they have an emulation technique that will accurately emulate CPU workload, and produce realistic results when collocating components of the SLICE scenario.

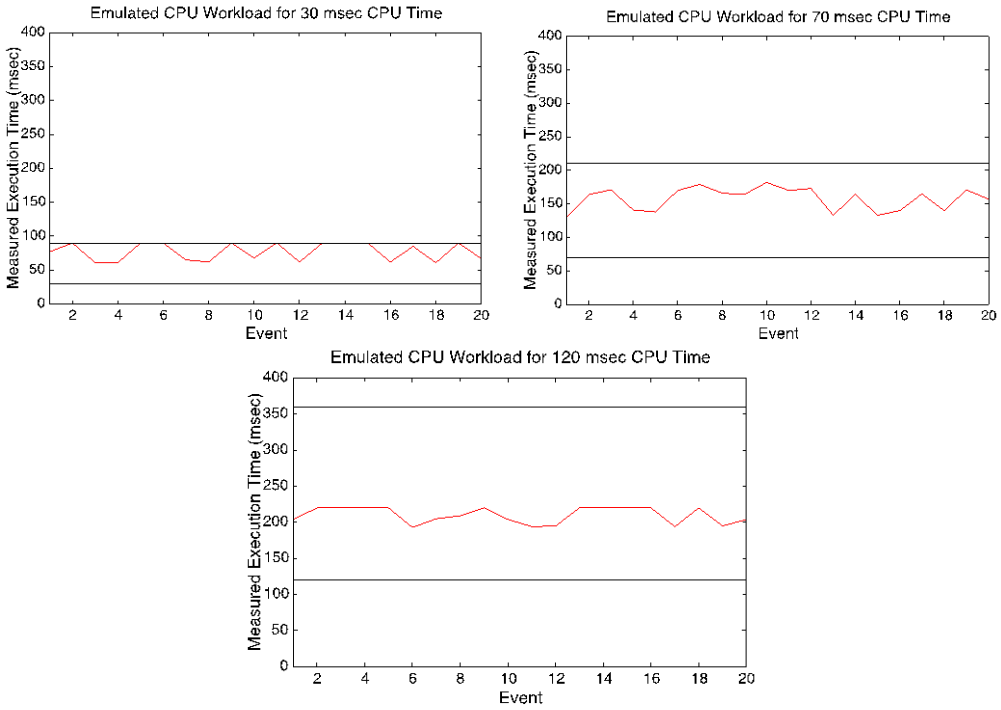


Figure 25 Measured CPU workload for three threads of execution on same processor.

4.4.4.2 Evaluating Performance of the SLICE Scenario

In Section 5.4.2, distributed system developers validated the emulation technique and capabilities of CUTS's CPU workload generator. Moreover, results showed that CUTS's CPU workload generator produces realistic behavior in environments with multiple threads of execution, such as collocating components of the SLICE scenario. Distributed system developers therefore plan to use the CUTS's CPU workload generator to evaluate the critical path of the SLICE scenario on different high-performance computing architectures.

In particular, distributed system developers plan to evaluate the improvement in performance of the critical path for the SLICE scenario when all components are collocated on the same host using different high-performance computing architectures. This will enable them to understand the side-effects of such architectures and can provide valuable insist in the future, such as determining how many hosts will be needed to ensure optimal performance of the SLICE scenario. They therefore used a single host in SI Lab to conduct several experiments.

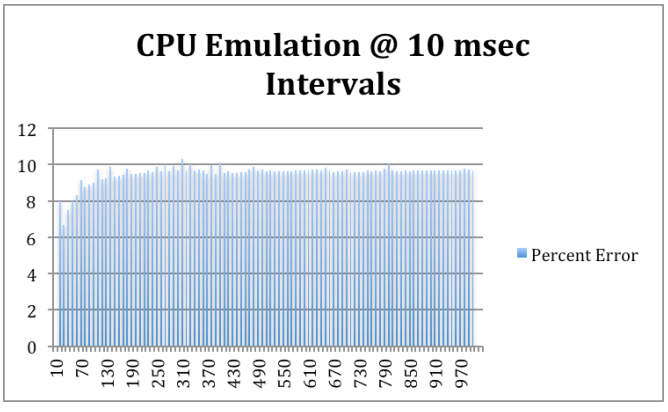
Table 6: Emulation results of SLICE scenario on different architectures.

Architecture	Avg. Exec. Time (msec)
Single	98.09
Dual-core	87.61
Dual-core (hyper-threaded)	65.8

Table 6 presents results of a single experiment that measured average end-to-end response time of the SLICE scenario's critical path. The experiment was executed on three different configurations/architectures of a single node is SI Lab. As highlighted in Table 6, the dual-core with hyper-threading had the best performance of the three, which distributed system developers of the SLICE scenario expected. More importantly, however, the results presented in Table 6 shows that CUTS's CPU workload generator enabled distributed system developers of the SLICE scenario to construct realistic experiments and observe the effects of different high-performance computing architecture configurations during early phases of the software lifecycle.

4.4.4.3 Applying Emulation Technique to Many-core Systems

We applied the same emulation approach on many-core systems (i.e., systems with more than 8 cores). Similar to how we calibrate the CPU workload generator in Section 5.4.1, we calibrate the CPU workload generator for a 64-core machine. Figure 26 shows the obtained validation results for the CPU workload generator at 10 msec intervals and 20 msec intervals against a single core in a 64-core machine.



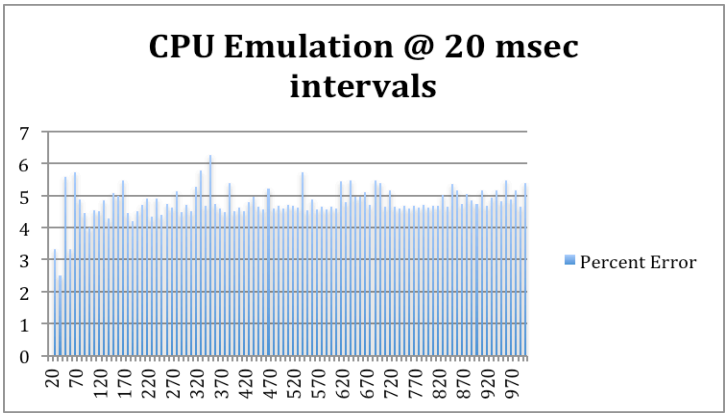
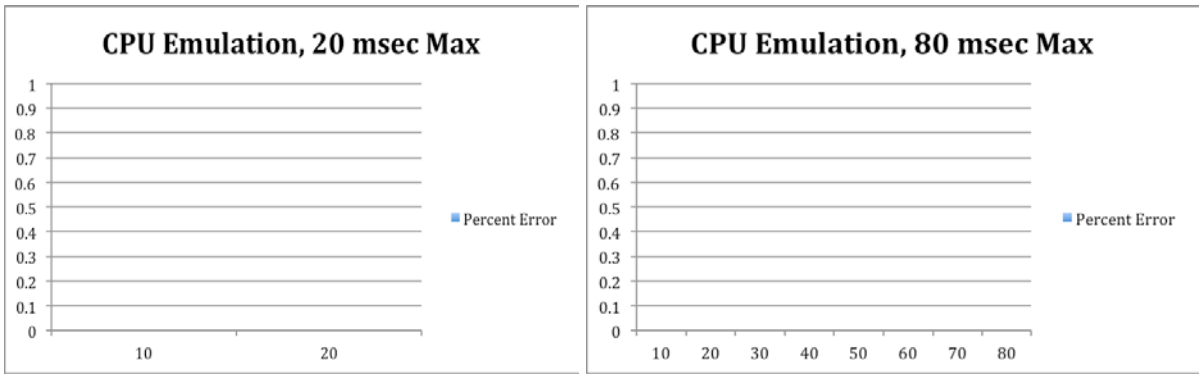


Figure 26 Validation of CPU workload generator on a single core in a 64-core machine.

As shown in the figures above, validation results of the CPU workload generator fails with a high percentage of error. The error eventually plateaus, but the percentage error between the emulated CPU workload generator execution time and the expected CPU execution time are well above 100%. After deeper investigation, we attribute the high error to the operating system working in the background to coordinate executing different processes/threads on a core.

We, however, noticed that for each interval we evaluated that the CPU workload generator was able to accurately emulate that account of CPU workload. For example, in Figure 26 the emulation results for 10 msec in the first graph, and 20 msec in the second graph are accurate. This is something we observed each time we attempt to validate the CPU workload generator at different intervals. Based on this behavior, we decided to select a max CPU execution time and validate the CPU workload generator by scaling it down instead of scaling the workload up. Figure 27 shows the results of calibrating the CPU workload generator with max CPU execution time of 20 msec, 80 msec, 640 msec, and 1000 msec.



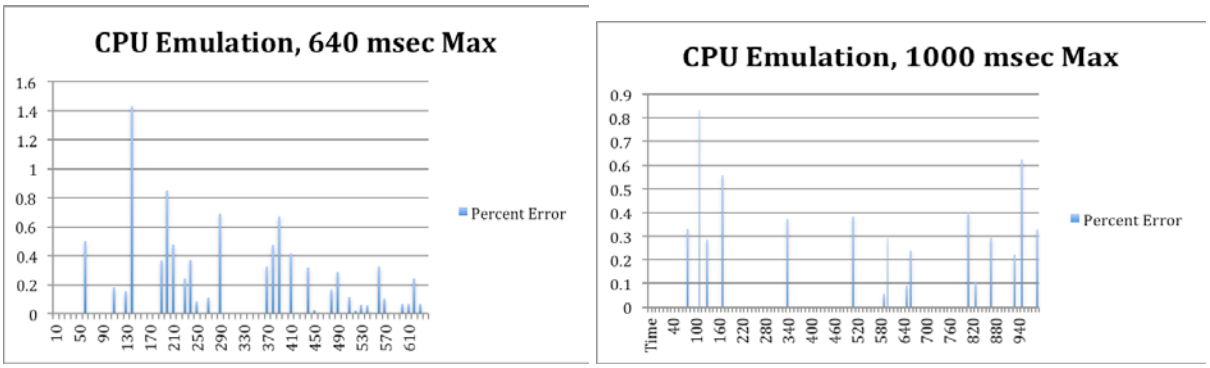


Figure 27 Validating the CPU workload generator at different max execution times, and scaling down CPU execution time to perform the validation.

As shown in Figure 27, validation results overall when choosing an max execution time for the CPU workload generator perform better at different execution times below the max execution time when compared to scaling a small interval to a larger execution time. Likewise, the larger the selected max CPU execution time, the greater the chance in having some intervals that have a high percentage error. We, however, learned that the execution times that experience high percent error is not deterministic. Finally, we observed that CPU execution times begin experiencing errors after setting 150 msec as the max CPU execution time. This, however, is not a problem for in our use cases since many of the CPU workloads for are less than this value. If you must execute CPU workloads for longer than 150 sec, then we recommend serially executing multiple CPU workload emulations to achieve the desired long execution time.

4.4.5 Related Work

MinneSPEC [94] and Biesbrouck et. al [80] present benchmark suites for generating CPU workload. It is designed to generate computation workload for new high-performance computing architectures being simulated. The CPU workload generation technique discussed in this article differs from MinneSPEC and Biesbrouck's work in that it abstracts away CPU characteristics that their benchmarks target in its computation workload. Moreover, the emulation technique discussed in this article is designed to evaluate application-level performance and MinneSPEC and Biesbrouck's work is designed to evaluate architecture-level performance. It believed, however, that the computation workload in MinneSPEC and Biesbrouck's work could be used as the abstract computation workload for the emulation technique presented in this article.

Leong et. al [93] present a technique for emulating CPU-bound workload in the context designing database system benchmarks. Their CPU workload emulation technique uses a simple arithmetic computation that is continuously executed based on the number of times specified in the benchmark. The emulation technique in this article extends their approach by representing similar arithmetic computations as abstract computations and accurately emulating them from [0,1000] msec. This enables distributed system developers to construct more controlled experiments when conducting system integration test during early phases of the software lifecycle.

4.4.6. Concluding Remarks and Lessons Learned

Evaluating enterprise DRE system performance on different high-performance computing architectures during early phases in the software lifecycle enables distributed system developers to make critical choices about system design and the target architecture before it is too late to change. Moreover, it enables them to identify performance bottlenecks before they become too costly to locate and rectify. This article therefore presented a technique for emulating computation intensive workload independent of the underlying high-performance computing architecture during early phases of the software lifecycle. The technique is based on abstracting CPU effects, such as fetching instructions from memory and CPU caching, and focusing primarily on overall execution time of a computation. Distributed system developers can therefore evaluate the performance of the system under development and make critical decision about system design and architecture choices in a timely manner. Based on the experience gained from applying the emulation technique for computation intensive enterprise DRE system, the following is a list of lessons learned:

- **Abstraction via emulation techniques simplifies construction of early integration test scenarios.** Instead of wrestling with low-level architecture concerns, distributed system developers focus on the overall execution times of the enterprise DRE system. This enables developers to create more realistic early system integration test with little effort, and concentrate more on evaluation the system under development on its target architecture.
- **Lack of details makes it hard to compare computation workload across heterogeneous architectures.** Different architectures have different characteristics, such as processor speed and type. Moving a computation from one architecture to another will yield different relative performance. Future work therefore includes enhancing the emulation techniques so computation intensive workloads have valid relative execution times when migrated between heterogeneous architectures.
- **Abstraction reduces the complexity of accurately emulating computation intensive workload.** This is because it does not rely on low-level, tedious, error-prone, and non-portable techniques, such as constantly sampling the clock and querying hardware (or software) for thread execution time.
- **Many-core machines require more upfront knowledge of workloads.** This is because we learned that you have to determine the maximum execution time of the CPU workload generator to enable proper calibration of the CPU workload generator under our approach.

CUTS and the CPU workload generator discussed in this article are available for download in open-source format at the following location: cuts.cs.iupui.edu.

5. CONCLUSIONS

Cyber-physical systems are computational systems with a close coupling between the computing and physical world. For example, cyber-physical DoD systems include, flight avionics, UAVs, and other platforms where the decisions made in software (e.g. how to move a plane's ailerons) have a direct impact on the physical world (e.g., whether or not the plane crashes). Moreover, the quality of service (QoS) aspects of how computation is performed (e.g. how fast, how securely) are just as important as the correctness of the computation (e.g. making the correct flight path decision too late could lead to a crash).

An open problem in the production of cyber-physical DoD systems involves predicting if the software architecture (including task parallelization/interaction and distribution of tasks to cores) will meet real-time deadlines and other QoS requirements in a given cyber-physical environment. It is hard to predict cyber-physical DoD system performance early in the lifecycle due to complex cache effects, unforeseen resource contention, physical dynamics, and layers of software abstractions coupled with a lack of open tools. As a result, changes to software architecture—such as changes to the task parallelization architecture and distribution of tasks to cores (e.g., to handle new requirements stemming from an expanded threat spectrum or in response to advances in multi-core hardware performance and scalability)—can be expensive if done late in the development cycle.

This proposed project tackles core research needed to automate and optimize the allocation of software to computing cores for cyber-physical DoD systems and create prototypical tools needed by DoD programs and developers. We propose to develop and prototype ***Cyber-physical multi-Core Optimization for Resource and cachE effectS (C^{2}ORES)***. C^{2}ORES is a system execution modeling and deployment optimization platform aimed at helping developers and testers of cyber-physical DoD systems choose a task parallelization and distribution architecture for a range of multi-core hardware platforms, including both homogeneous and heterogeneous multi-core hardware platforms.

This research was conducted by a collaborative team from Vanderbilt University, Virginia Tech and Indiana University-Purdue University at Indianapolis. Within the one year of this project, we were able to address some of the vast set of challenges that exist in this space. Many more challenges exist. For example, how can multicore scaling including cache optimizations work in a distributed context where not only local concerns but more holistic concerns play a key role. Second, how can this research be carried forward when resilience of the system is an important concern. Our future research will investigate many of these unresolved challenges.

6 REFERENCES

[1] T. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In Proceedings of the 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS05), pages 213–223, 2005.

[2] C. Fonseca, P. Fleming, et al. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In Proceedings of the fifth international conference on genetic algorithms, pages 416–423. Citeseer, 1993.

[3] G. Anastasi, A. Falchi, A. Passarella, M. Conti, and E. Gregori. Performance measurements of motes sensor networks. In Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems, pages 174–181. ACM New York, NY, USA, 2004.

[4] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS), pages 113–121, 2003.

[5] T. Ba¨ck. Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Oxford University Press, USA, 1996.

[6] H. Beitollahi and G. Deconinck. Fault-Tolerant Partitioning Scheduling Algorithms in Real-Time Multiprocessor Systems. Pacific Rim International Symposium on Dependable Computing, IEEE, 0:296–304, 2006.

[7] A. Bertossi, L. Mancini, and F. Rossini. Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems. IEEE Transactions On Parallel and Distributed Systems, pages 934–945, 1999.

[8] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. New Strategies for Assigning Real-time Tasks to Multiprocessor Systems. IEEE Transactions on Computers, 44(12):1429–1442, 1995.

[9] A. Carzaniga, A. Fuggetta, S. Richard, D. Heimbigner, A. van der Hoek, A. Wolf, A Characterization Framework for Software Deployment Technologies. Defense Technical Information Center, 1998.

[10] S. Lauzac, R. Melhem, and D. Mosse. Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor. In 10th Euromicro Workshop on Real Time Systems, pages 188–195, 1998.

[11] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. Swarm Intelligence, 1(1):33–57, 2007.

[12] N. R. C. Steering Committee for the Decadal Survey of Civil Aeronautics. Decadal Survey of Civil Aeronautics: Foundation for the Future. The National Academies Press, 2996.

[13] J. White, B. Dougherty, C. Thompson, D. C. Schmidt, ScatterD: Spatial Deployment Optimization with Hybrid Heuristic / Evolutionary Algorithms, ACM Transactions on Autonomous and Adaptive Systems Special Issue on Spatial Computing, (to appear). This research has been funded in part by NSF Award #CNS 0915976.

[14] J. White, B. Doughtery, D. C. Schmidt, ASCENT: An Algorithmic Technique for Designing Hardware and Software in Tandem, IEEE Transactions on Software Engineering Special Issue on Search-based Software Engineering, (to appear). This research has been funded in part by NSF Award #CNS 0915976.

[15] J. White, D. C. Schmidt, Automating Deployment Planning with an Aspect Weaver, IET Software Special Issue on Domain-specific Modeling Languages for Aspect-Oriented Programming, Volume 3, Issue 3 , p. 167-183, June 2009

[16] J. White, Douglas C. Schmidt, Andrey Nechypurenko, Egon Wuchner, Model Intelligence: an Approach to Modeling Guidance, UPGRADE Journal, Volume 9, Number 2, pgs. 22-28, April 2008

[17] J. White, J. Hill, J. Gray, S. Tambe, D. C. Schmidt, A. Gokhale, Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques, IEEE Software Special Issue on Domain-Specific Languages and Modeling July/August, 2009, Volume 26, Number 4, pgs. 47-53

[18] J. White and Douglas C. Schmidt, Model-Driven Product-Line Architectures for Mobile Devices, Proceedings of the 17th Annual Conference of the International Federation of Automatic Control, 6pgs., July 6-11, 2008, Seoul, Korea

[19] J. White and D. C. Schmidt, Automated Configuration of Component-based Distributed Real-time and Embedded Systems from Feature Models, Proceedings of the 17th Annual Conference of the International Federation of Automatic Control, 6pgs., July 6-11, 2008, Seoul, Korea

[20] J. White, K. Czarnecki, D. C. Schmidt, G. Lenz, C. Wienands, E. Wuchner, and L. Fiege, Automated Model-based Configuration of Enterprise Java Applications, Enterprise Computing Conference (EDOC), 12pgs., October, 2007, Annapolis, Maryland

[21] J. White, D. Schmidt, and A. Gokhale, Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study, Proceedings of MODELS 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, 11pgs., Half Moon Resort, Montego Bay, Jamaica, October 5-7, 2005. (Selected as a best paper)

[22] J. White, D. Schmidt, and A. Gokhale, The J3 Process for Building Autonomic Enterprise Java Bean Systems, Proceedings of the International Conference on Autonomic Computing (ICAC 2005), 2pgs., Seattle, WA, June 2005

[23] J. White, D. Benavides, B. Dougherty, D. C. Schmidt, Automated Reasoning for Multi-step Software Product-line Configuration Problems, Software Product-lines Conference (SPLC), 10pgs. August 24-28, 2009, San Francisco, CA

[24] B. Dougherty, J. White, J. Balasubramanian, C. Thompson, and D. C. Schmidt, Deployment Automation with BLITZ, 31st International Conference on Software Engineering, May 16-24, 2009 Vancouver, Canada.

[25] J. White, Douglas C. Schmidt, D. Benavides, P. Trinidad, A. Ruiz-Cortez, Automated Diagnosis of Product-line Configuration Errors in Feature Models, Software Product Lines Conference (SPLC), 10pgs., September, 2008, Limmerick, Ireland

[26] B. Dougherty, J. White, C. Thompson, and D. C. Schmidt, Automating Hardware and Software Evolution Analysis, 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), April 13-16, 2009 San Francisco, CA USA.

[27] H R Boveiri. ACO-MTS: A new approach for multiprocessor task scheduling based on ant colony optimization. pages 1–5, 2010.

[28] T D Braun, H J Siegel, N Beck, L L Bölöni, M Maheswaran, A I Reuther, J P Robertson, M D Theys, B Yao, and D Hensgen. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[29] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[30] Hua Chen, Albert Mo Kim Cheng, and Ying-Wei Kuo. Assigning real-time tasks to heterogeneous processors by applying ant colony optimization. *Journal of Parallel and Distributed Computing*, 71(1):132–142, January 2011.

[31] Hua Chen, Albert Mo Kim Cheng, and Ying-Wei Kuo. Assigning real-time tasks to heterogeneous processors by applying ant colony optimization. *Journal of Parallel and Distributed Computing*, 71(1):132–142, January 2011.

[32] P Chretienne, E G Coffman, J K Lenstra, Z Liu, and P Brucker. *Scheduling theory and its applications*, volume 149. John Wiley & Sons, 1995.

[33] Luis Pedro Coelho. Jug: A task-based parallelization framework. http://luispedro.org/software/jug, November 2009.

[34] A. Colorni, M. Dorigo, V. Maniezzo, et al. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Paris, France, 1991.

[35] D Cordeiro, G Mounié, S Perarnau, D Trystram, J M Vincent, and F Wagner. Random graph generation for scheduling simulations. page 60, 2010.

[36] Robert P Dick, David L Rhodes, and Wayne Wolf. TGFF: task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*. IEEE Computer Society, March 1998.

[37] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[38] Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(6):911–924.

[39] M. R. Gary and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness. 1979.

[40] N G Hall and M E Posner. Generating experimental data for computational testing with machine scheduling applications. *Operations Research*, 49(6):854–865, 2001.

[41] T Hammerl. Ant colony optimization for tree and hypertree decompositions. *Master's Thesis, Vienna University of Technology*, 2009.

[42] E.S.H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 5(2):113–120, 1994.

[43] H Jin, H Wang, H Wang, and G Dai. An ACO-Based approach for task assignment and scheduling of multiprocessor control systems. *Theory and Applications of Models of Computation*, pages 138–147, 2006.

[44] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 33(11):1023–1029, 1984.

[45] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[46] Y.K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[47] J K Lenstra and A H G R Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[48] Shih-Tang Lo, Ruey-Maw Chen, Yueh-Min Huang, and Chung-Lun Wu. Multiprocessor system scheduling with precedence and resource constraints using an enhanced ant colony system. *Expert Systems with Applications*, 34(3):2071–2081, April 2008.

[49] G Ritchie. Static multi-processor scheduling with ant colony optimisation & local search. *Master of Science thesis, University of Edinburgh*, pages 1–101, 2003.

[50] G U Srikanth, V U Maheswari, and A P Shanthi. Tasks Scheduling Using Ant Colony Optimization. *Journal of Computer …*, July 2012.

[51] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[52] "Documentation of verification, validation, &accreditation for models and simulations," 2009, doDI 5000.61.

[53] A.-H. Liu and R. P. Dick, "Automatic run-time extraction of communication graphs from multithreaded applications," pp. 46–51, 2006.

[54] K. S. Vallerio and N. K. Jha, "Task graph extraction for embedded system synthesis," pp. 480–486, 2003.

[55] R. Ha and J. W. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," pp. 162–171, 1994.

[56] T. M. Takai, "Cloud computing strategy," DTIC Document, Tech. Rep., 2012.

[57] A. Corradi, L. Foschini, J. Povedano-Molina, and J. M. Lopez-Soler, "Dds-enabled cloud management support for fast task offloading," in *Computers and Communications (ISCC), 2012 IEEE Symposium on*, EEE, 2012, pp. 000 067–000 074.

[58] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. plus 0.5em minus 0.4emUSENIX Association, 2005, pp. 273–286.

[59] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2008, pp. 161–174.

[60] Y. Du and H. Yu, "Paratus: Instantaneous failover via virtual machine replication," in *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*. plus 0.5em minus 0.4emIEEE, 2009, pp. 307–312.

[61] Y. Tamura, K. Sato, S. Kihara, and S. Moriai, "Kemari: Virtual machine synchronization for fault tolerance," *In USENIX 2008 Poster Session*, 2008.

[62] K. An, F. Caglar, S. Shekhar, and A. Gokhale, "Automated Placement of Virtual Machine Replicas to Support Reliable Distributed Real-time and Embedded Systems in the Cloud," in *International Workshop on Real-time and Distributed Computing in Emerging Applications (REACTION), 33rd IEEE Real-time Systems Symposium (RTSS '12)*. San Juan, Puerto Rico, USA: IEEE, Dec. 2012.

[63] K. An, S. Shekhar, F. Caglar, A. Gokhale, and S. Sastry, "A Cloud Middleware for Assuring Performance and High Availability of Soft Real-time Applications," *Submitted to Special Issue of REACTION 2012 Workshop, Elsevier Journal of Systems Architecture (JSA)*, 2014.

[64] F. Caglar and A. Gokhale, "iOverbook: Managing Cloud-based Soft Real-time Applications in a Resource-Overbooked Data Center," in *Submitted to the 20th IEEE Real-time and Embedded Technology and Applications Symposium*. Berlin, Germany: IEEE, Apr. 2014.

[65] S. A. Baset, L. Wang, and C. Tang, "Towards an understanding of oversubscription in cloud," in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, USENIX Association, 2012, pp. 7–7.

[66] (2013, Sep.) Best practices for oversubscription of cpu, memory and storage in vsphere virtual environments. [Online]. Available: https://software.dell.com =0pt

[67] F. Caglar, S. Shekhar, and A. Gokhale, "A Performance Interference-aware Virtual Machine Placement Strategy for Supporting Soft Real-time Applications in the Cloud," Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA, Tech. Rep. ISIS-13-105, 2013.

[68] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[69] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: challenges and approaches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 3, pp. 55–60, 2010.

[70] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden, “Miyakodori: A memory reusing mechanism for dynamic vm consolidation,” in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 606–613.

[71] U. Deshpande, X. Wang, and K. Gopalan, “Live gang migration of virtual machines,” in *Proceedings of the 20th international symposium on High performance distributed computing*. plus 0.5em minus 0.4em ACM, 2011, pp. 135–146.

[72] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi, “Making vm consolidation more energy-efficient by postcopy live migration,” in *CLOUD COMPUTING 2011, The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011, pp. 195–204.

[73] “Reactive consolidation of virtual machines enabled by postcopy live migration,” in *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*. ACM, 2011, pp. 11–18.

[74] J. L. Berral, Í. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, and J. Torres, “Towards energy-aware scheduling in data centers using machine learning,” in *Proceedings of the 1st International Conference on energy-Efficient Computing and Networking*ACM, 2010, pp. 215–224.

[75] J. T. Piao and J. Yan, “A network-aware virtual machine placement and migration approach in cloud computing,” in *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*. IEEE, 2010, pp. 87–92.

[76] A. Khosravi, S. K. Garg, and R. Buyya, “Energy and carbon-efficient placement of virtual machines in distributed cloud data centers,” in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 317–328.

[77] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, and S. Cheng, “Energy-saving virtual machine placement in cloud data centers,” in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 618–624.

[78] J. J. More, “The Levenberg-Marquardt Algorithm: Implementation and Theory,” in *Numerical Analysis*, ser. Lecture Notes in Mathematics, G. Watson, Eds, Springer Berlin Heidelberg, 1978, vol. 630, pp. 105–116. [Online]. Available: http://dx.doi.org/10.1007/BFb0067700 =0pt

[79] C. Reiss, J. Wilkes, and J. Hellerstein, “Google cluster-usage traces: format+ schema,” *Google Inc., White Paper*, 2011.

[80] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Representative Multiprogram Workloads for Multithreaded Processor Simulation. In *IEEE 10th International Symposium on Workload Characterization*, pages 193–203, September 2007.

[81] S. Bohacek, J. Hespanha, J. Lee, and K. Obraczka. A hybrid systems modeling framework for fast and accurate simulation of data communication networks. In *Proceedings of ACM SIGMETRICS ’03*, June 2003.

[82] J.T.Buck,S.Ha,E.A.Lee,andD.G.Messerschmitt.Ptolemy:AFrameworkforSimulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies*, 4, Apr. 1994.

[83] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383,

Aug. 2001.

[84] G. de A Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *Computers, IEEE Transactions on*, 52(10):1332–1346, Oct. 2003.

[85] A. Haghighat and M. Nikravan. A Hybrid Genetic Algorithm for Process Scheduling in Distributed Operating Systems Considering Load Balancing. In *In Proceedings of Parallel and Distributed Computing and Networks*, February 2005.

[86] M.Hauswirth,A.Diwan,P.F.Sweeney,andM.C.Mozer.AutomatingVerticalProfiling.In *Proceedings of the* 20th *Annual ACM SIGPLAN Conference on Object-Oriented Program- ming, Systems, Languages, and Applications (OOPSLA 05)*, pages 281–296, New York, NY, USA, 2005. ACM Press.

[87] J. H. Hill and A. Gokhale. Model-driven Engineering for Early QoS Validation of Component-based Software Systems. *Journal of Software (JSW)*, 2(3):9–18, Sept. 2007.

[88] J. H. Hill and A. Gokhale. Model-driven Specification of Component-based Distributed Real-time and Embedded Systems for Verification of Systemic QoS Properties. In *Proceeding of the Workshop on Parallel, Distributed, and Real-Time Systems (WPDRTS '08)*, Miami, FL, April 2008.

[89] J.H.Hill and A.Gokhale. Towards Improving End-to-End Performance of Distributed Real-time and Embedded Systems using Baseline Profiles. *Software Engineering Research, Management and Applications (SERA 08), Special Issue of Springer Journal of Studies in Computational Intelligence*, 150(14):43–57, Aug. 2008.

[90] J.H.Hill, J.Slaby, S.Baker, and D.C.Schmidt. Applying System Execution ModelingTools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[91] M. D. Hill. Opportunities Beyond Single-core Microprocessors. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 97– 97, New York, NY, USA, 2008. ACM.

[92] S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jun 2006.

[93] H. J. Jeong and S. H. Lee. A Workload Generator for Database System Benchmarks. In *Proceedings of the 7th International Conference on Information Integration and Web-based Applications & Services*, pages 813–822, September 2005.

[94] A. Klein Osowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters*, 1(1):7, 2002.

[95] A´.Le´deczi,A´.Bakay, M.Maro´ti, P.Vo¨lgyesi, G.Nordstrom, J.Sprinkle, and G.Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[96] M. W. Masters and L. R. Welch. Challenges For Building Complex Real-time Computing Systems. *Scientific International Journal for Parallel and Distributed Computing*, 4(2), 2001.

[97] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[98] Rittel, H. and Webber, M. Dilemmas in a General Theory of Planning. *Policy Sciences*, pages 155–169, 1973.

[99] C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, Boston, MA, USA, September 2001.

[100] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

List of Acronyms

Ant colony optimization (ACO)

Ant Colony Optimization (ACO)

Ant Colony System (ACS)

artificial neural network (ANN)

Cloud service providers (CSPs)

Component Utilization Test Suite (CUTS)

Cyber-physical multi-Core Optimization for Resource and cachE effectS (C^2ORES)

Deployment Optimization Validation Engine (DOVE)

Directed Acyclic Graph (DAG)

distributed real-time and embedded

high performance computing (HPC)

mean squared error (MSE)

model-driven engineering (MDE)

multi-core deployment optimization (MCDO)

Multi-core Deployment Optimization (MCDO)

network file system (nfs)

Service Level Agreements (SLAs)

Simulated Annealing (SA)

Standard Task Graph (STG) Systems and Software PRoducibility

CollaborationExperimentation Environment (SPRUCE)

virtual machine monitor (VMM)

virtual machines (VMs)